Tool Support for Assurance Case Development

Ewen Denney · **Ganesh Pai**

Abstract Argument-based *assurance cases*, often represented and organized using graphical *argument structures*, are increasingly being used in practice to provide assurance to stakeholders, e.g., regulatory authorities, that a system is acceptable for its intended use with respect to dependability and safety concerns. In general, comprehensive system-wide assurance arguments aggregate a substantial amount of diverse information, such as the results of safety analysis, requirements analysis, design, verification and other engineering activities. Although a variety of assurance case tools exist, many desirable operations on argument structures such as hierarchical and modular abstraction, argument pattern instantiation, and inclusion/extraction of richly structured information have limited to no automation support. To close this automation gap, over the past four years we have been developing a toolset for assurance case automation, AdvoCATE, at the NASA Ames Research Center. This paper describes how AdvoCATE is being engineered atop formal foundations for assurance case argument structures, to provide unique capabilities for: *a*) automated creation and assembly of assurance arguments, *b*) integration of formal methods into wider assurance arguments, *c*) automated pattern instantiation, *d*) hierarchical abstraction, *e*) queries and views, and *f*) verification of arguments. We (and our colleagues) have used AdvoCATE in real projects for safety assurance, in the context of unmanned aircraft systems.

Keywords Assurance Cases · Safety Cases · Automation · Tool Support · Formal Methods

This work has been supported, in part, by the Safe Autonomous Systems Operations (SASO) project of the Airspace Operations and Safety Program, of the NASA Aeronautics Research Mission Directorate.

E. Denney
SGT / NASA Ames Research Center, Moffett Field, CA, USA
E-mail: ewen.denney@nasa.gov

G. Pai
SGT / NASA Ames Research Center, Moffett Field, CA, USA
E-mail: ganesh.pai@nasa.gov

1 Introduction

The submission of a *safety assurance case* (or *safety case* for short) is increasingly being required during system certification in many safety-critical industries, such as aviation [33], nuclear power [47], road[1] and rail transportation [49, 74], defense [72], and medical devices [78]. Essentially, a safety case provides an audit trail of safety considerations, from concept through operations, that can assist in convincing the various stakeholders of a system, including regulators, that the system is *acceptably safe* [73]. The purpose is, broadly, to demonstrate that

- the risks associated with a specific system concern, such as safety or security, have been identified, are well-understood, and have been appropriately controlled; and
- there are processes in place to monitor the performance, and effectiveness of risk controls.

The use of *assurance cases*[2] has also been recommended during procurement and mission assurance of aerospace systems [31].

Generally speaking, a safety case is a comprehensive, defensible, and valid justification of the safety of a system for a given application in a defined operating environment. Often, the core of this justification is a structured, risk-based argument[3], which links safety-related *claims* through a *chain of reasoning* to a body of the appropriate *evidence*. One of the motivations to use structured arguments is to explicitly capture the traceability between safety claims and the substantiating evidence. Another motivation is to make it easier to understand and critically review the safety case [44]. To further improve clarity, graphical notations have emerged over the past decade to present the elements of a safety argument, e.g., the Claims-Argument-Evidence (CAE) notation [9], and the Goal Structuring Notation (GSN) [39]. Such graphical *argument structures*, can be thought of as a visual index into the safety reasoning and evidence comprising a safety case.[4]

Existing tools[5] that support the development of safety arguments using the GSN or CAE [1, 4, 53, 56, 66] largely facilitate a *manual* development of safety cases. That is, in practice, safety case authors need to specify the arguments that connect low-level safety evidence to higher-level safety claims, and manually perform operations such as argument pattern instantiation, e.g., using *copy-and-paste*. Additionally, to our knowledge, existing tools lack a formal basis—partly due to the informality of the notations themselves—which, we believe, has impeded the development of certain types of automation capabilities. In particular, operations such as hierarchical and modular abstraction, pattern instantiation, inclusion and extraction of richly structured information, and evaluation of argument properties, can be usefully auto-

[1] Strictly speaking, road vehicles do not undergo regulatory certification in the same way as civil aircraft; rather, they are *qualified* by the manufacturer as meeting an applicable safety standard.

[2] In general, an assurance case provides assurance of broad system concerns, such as dependability, safety, and security; a safety case is a specialization of an assurance case for system safety assurance.

[3] There are also different (but compatible) notions of safety case [6, 76] in which there is no explicit requirement for presenting structured arguments.

[4] In this paper, we will use the terms *safety argument*, and *safety case* interchangeably when the distinction between the two is not significant. Also note that the scope of our work here applies to assurance cases in general, although we will focus primarily on safety assurance.

[5] Also see Section 7.1 for more details on existing tools.

mated to support the development of large safety arguments, typical of real systems. Over the past four years we have been developing the Assurance Case Automation Toolset (AdvoCATE), atop formal foundations, to close this automation gap. In this paper, we describe how we are engineering AdvoCATE to provide unique capabilities for automating, to the extent possible: *a*) the creation and assembly of safety assurance arguments, *b*) the integration of formal methods into wider assurance arguments, *c*) instantiation of assurance *argument patterns*,[6] *d*) hierarchical and modular abstraction, *e*) the generation of *views* from user-defined *queries*, and *f*) the verification of properties of argument structures.

Our hypothesis is that automation will not only assist in reducing the time and effort spent in creating, understanding, evaluating, and managing argument structures, but also reduce the potential to create ill-formed/inadequate arguments. Thus we believe that automation support can address the practical challenges that we have encountered in assurance case development. The main contributions of this paper are: *i*) developing and integrating novel automation capabilities (as identified above), based on formal foundations, into a single toolset; and *ii*) elaborating a methodology for assurance argument development that leverages our toolset and its automation features.

The rest of this paper is organized as follows: Section 2 first sets the context for using AdvoCATE, describing a process for safety argument development, after which Section 3 motivates the needs for automation support during that process, identifying the broad requirements for the relevant capabilities. Then, in Section 4, we summarize the foundations for the required automation features providing a formal, abstract specification for their implementation, which we present in Section 5. This section also describes the AdvoCATE system architecture. Thereafter, in Section 6, we give an overview of the basic functionality of our toolset, after which we illustrate how its automation capabilities were used during the creation of safety arguments for real aviation systems. In Section 7, first we discuss related work, contrasting the capabilities of existing tools with the innovations described in this paper. Thereafter, we identify opportunities for future work. Section 8 concludes the paper.

2 Methodology

In this section, first we present our vision of a safety case. Then we describe an abstract safety assurance methodology and our process for safety case development, which will provide the context for how AdvoCATE will be used, and its automation requirements (Section 3).

2.1 Overview

An argument is a connected series of propositions used in support of the truth of an overall proposition. The latter is usually referred to as a *claim*, whereas the former represents a chain of reasoning connecting the claim and the *evidence*. Our vision

[6] In the rest of the paper, we will use *pattern* to mean an argument pattern. Also see Section 4, and [20].

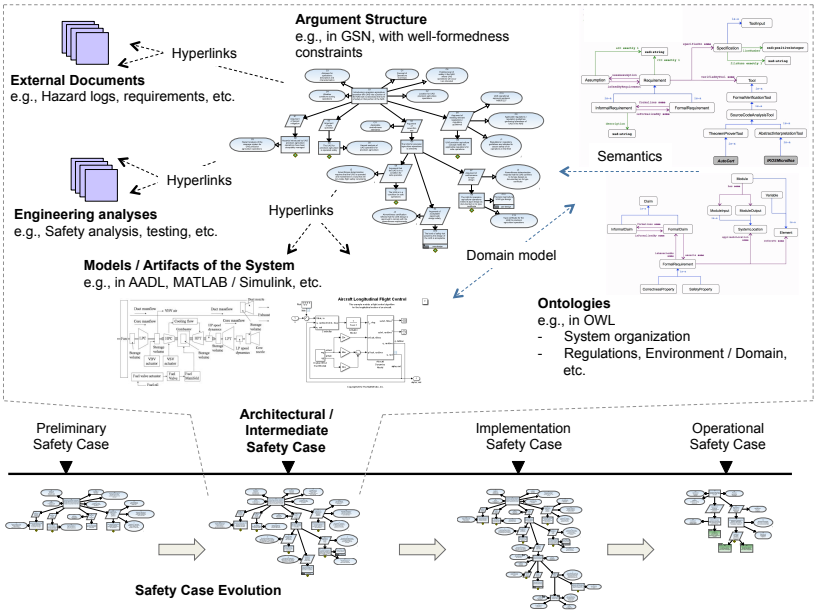


Fig. 1. Our vision of a safety case as an evolving risk management artifact, whose skeleton is an argument structure assimilating and linking diverse reasoning and evidence.

of a safety case (Fig. 1) is a structured and evolving argument that comprises explicit safety claims, assimilates heterogeneous evidence, and presents the reasoning required to conclude[7] that a system will be safe for a defined application and operating environment. The elements of the safety case are given as an *argument structure*, i.e., a diagrammatic presentation of the argument using the Goal Structuring Notation (GSN)[8]. The nodes of the argument structure contain links to external items including

- artifacts such as hazard logs, requirements documents, design documents, various relevant models of the system, etc.;
- the results of engineering activities, e.g., safety, system, and software analyses, various inspections, reviews, simulations, and verification activities including different kinds of system, subsystem, and component-level testing, formal verification, etc.;
- records from ongoing operations, as well as prior operations, if applicable; and, additionally,
- nodes containing *metadata* drawn from domain ontologies that provide supplementary and relevant domain-specific semantic information.

[7] Or, convince and communicate to the relevant stakeholders.

[8] See Section 4 for more details. We have opted to use the GSN, although other appropriate notations could also have been used.

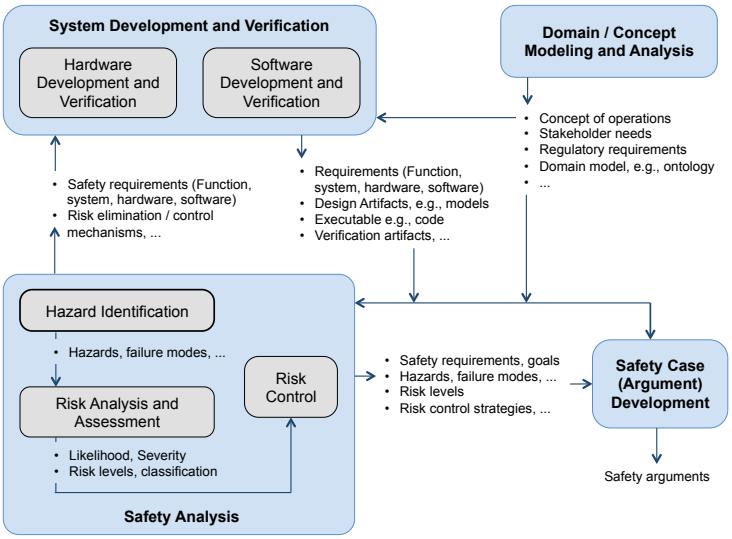


Fig. 2. Abstract safety assurance methodology showing data flow between the processes for domain modeling and analysis, system development and verification, safety analysis and safety argument development.

Fig. 1 illustrates the evolution of a safety case and the underlying argument structures. During the early phases of system development, we create a *preliminary* safety case that addresses the safety aspects of system concepts, e.g., by identifying high-level safety goals. Thereafter, we develop an *intermediate* and/or *architectural* safety case to reason about risk mitigation through system design, as well as to address the trade-offs between safety and other competing design concerns, such as performance, reliability, etc. *Implementation* and *operational* safety cases are then created before deployment, and updated during system operations, respectively.

Each of these safety cases is self-contained, although interrelated. Together they represent the evolution of system safety concerns from concept through operations. To create the corresponding argument structures, we apply our process for safety argument development, which we will describe subsequently (Section 2.3).

2.2 Safety Assurance

Fig. 2 shows an abstract and simplified (data flow) view of a safety assurance methodology[9], that utilizes a safety argument development process. In particular, the pro-

[9] We note that the data given in Fig. 2 is not comprehensive. In actual practice, this abstract methodology is replaced by concrete processes, activities, and the corresponding data, e.g., as recommended in civil aviation guidelines for system development and safety assessment [64, 65]. Additionally, we note that this methodology addresses safety assurance *prior* to system operations, and is applied towards facilitating the decision to release a system into service. A lifecycle approach to safety assurance [16] also takes into account operational safety measures and safety performance, although we will not address that here.

cesses of domain modeling and analysis, system development and verification, and safety analysis provide the core data for safety argument development, whereas the management activities of the methodology (not shown here) define the plan for safety management.

For example, through domain modeling and analysis we determine stakeholder needs, formulate the scope of the technical problem, and define the operating context for the engineered system. Among the outcomes of this process is a domain model, e.g., in the form of an ontology, from which metadata can be drawn for the nodes of the argument structure. Similarly, the processes of safety analysis, system development and verification, are sources of product-specific assurance claims, reasoning, and evidence from which the safety arguments will be built. In addition, the processes are also sources of the artifacts, results, and records to which the nodes of the argument structures will be linked (as described earlier in Section 2.1). We perform safety analysis both during concept development and subsequently during system development. As development proceeds, we refine the system design, together with its safety analysis and the associated assurance argument. After implementation, and the generation of verification evidence, we revisit the safety analysis to ascertain that the safety requirements have been satisfied, following which the items of evidence are assimilated into the assurance argument.

Note that, in Fig. 2, the safety analysis, system development and verification, and safety argument development processes are concurrent and iterative, and are are periodically synchronized, e.g., at the milestones defined during the plan for system development and certification [17]. We have intentionally not described a workflow of the processes, since we can define a number of possible realizations based on the constraints of the domain and projects.

2.3 Safety Argument Development

Fig. 3 shows our process for safety argument development identifying six key activities[10]; our main focus is on the process activities and the associated data flow instead of a control flow, since the order in which the activities are performed must be tailored to the specific needs of the project/domain. In general, however, argument development can be performed in a *top-down*, or a *bottom-up*, manner. The former requires a definition of a high-level argument organization and/or claim, followed by successive refinement into lower-level details. The latter, in contrast, is concerned with assembly of an argument based on the inferences that can be drawn from the available and existing lower-level details (i.e., the evidence).

At each milestone, the process for safety argument development produces an argument structure that reflects: *i*) the inclusion of specific artifacts available at a given phase of the system development process, and *ii*) the evolving state of the safety argument (e.g., as shown in Fig. 1) and, consequently, system safety.

Note that safety argument evolution as shown in Fig. 1 is coarse-grained, although we can define a finer-grained evolution. Thus, the architectural safety case, for exam-

[10] The process shown in Fig. 3 is different from (but compatible with) both the *six-step method* for developing an argument structure [39], and the safety case development methodology in [8].

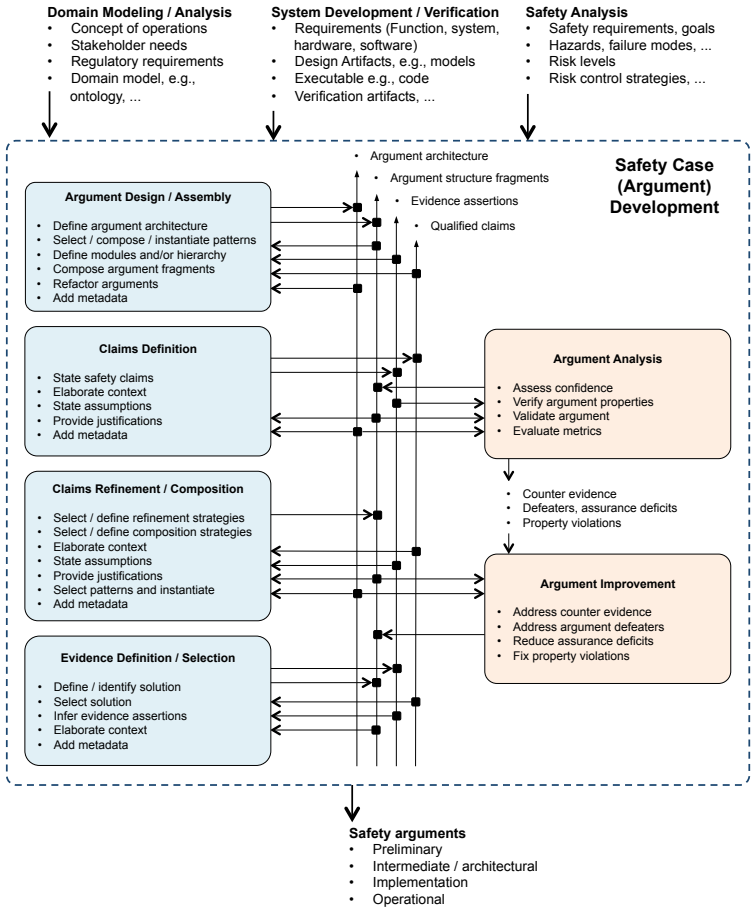


Fig. 3. Safety argument development process showing the key activities and the data flow; the data produced by the remaining processes of the safety assurance methodology (Fig. 2), are consumed by the argument development process, and the result is a series of argument structures (Fig. 1) which embody the evolution of the system safety case.

ple, can be considered to be the result of assembling the argument elements produced at the various milestones that correspond to the phases of system design, and system architecture development.

2.3.1 Argument Design / Assembly

In a top-down development of safety arguments, we define an *argument architecture* which specifies a high-level, abstract, organization of the overall structure and

elements of the argument [24]. Specifically, we select and compose *argument patterns*[11], i.e., abstractions representing various styles of argument, taking into account system assurance concerns, the types of claims requiring support, and *argument design criteria* such as compliance with safety principles, reducing the cost of re-certification, modular organization, maintainability, etc. We instantiate patterns using domain- and system-specific data (which may, themselves, be generated using a tool, e.g., for formal verification), to produce fragments of *instance arguments*. We can directly compose instance arguments (to be consistent with pattern composition), or there may be a need to introduce manually-created, intermediate, '*glue* arguments'.

Based on the project needs, we can specify a modular organization of the argument architecture (see Section 6.3.1), which can be beneficial in a number of ways, e.g., to reflect the modularity inherent in the system, to manage safety argument size, to constrain the impact of changes during argument evolution, as well as to support distributed development. Additionally, we can also introduce a hierarchical organization in argument fragments (see Section 6.3.2) to reflect the natural hierarchy of claims, or the refinement of an abstract argument fragment into a more detailed argument, e.g., when creating *evidence arguments* [21].

During both top-down, and *bottom-up development*, we use the argument fragments produced from pattern instantiation, along with those produced from the remaining activities (e.g., as shown in Fig. 3), claims refinement/composition, and evidence definition/selection) to assemble an argument consistent with its architecture. As both the system and its assurance argument evolves, we refactor the argument to improve its comprehensibility, and the consistency with its architecture and argument design criteria, such as maintainability. Essentially, the activity of argument design/assembly is one wherein we combine top-down and bottom-up argument development, together with tasks for improving its understandability and supporting its evolution.

2.3.2 Claims Definition

We define safety claims based upon the system development phase, and the available artifacts, e.g., using safety requirements identified through hazard analyses. We additionally state the context in which the claims can be interpreted and determined to be valid, together with any relevant assumptions and the necessary justifications. Note that assumptions can be made both about the system for which assurance is sought, and the environment in which the system operates.[12] We will refer to the combination of claim, its associated context, justifications and assumptions as a *qualified claim*.

For example, in a safety argument, a correctness claim about a software function should be accompanied with its specification as context, an assumption that the specification is valid, and a justification of the bearing of the correctness of that function on system safety. The results of claims definition are propositions—specified at an appropriate level of abstraction—concerning system, subsystem and component properties, which have been determined through safety analysis to have a (direct or

[11] For more details on argument patterns, see Section 4.2.

[12] Additionally, assumptions can also be made about the assurance techniques employed.

indirect) bearing on system safety, e.g., the reliability of a hardware fail-safe, the correctness of a software switch, etc. When a claim concerns low-level assertions about evidence items, we distinguish them as *evidence assertions*. Thus, an evidence assertion is a claim (in a goal node) that immediately precedes the solution node corresponding to the evidence item.

2.3.3 Claims Refinement / Composition

The core task of this activity is to define and/or select the appropriate strategies to link related claims. Additionally, we specify the associated rationale, i.e., the appropriate assumptions and justifications in which strategies can be reasonably used, as well as the relevant context.

Specifically, claims refinement is performed in a top-down development of arguments. Here, we successively decompose higher-level, abstract claims into progressively more detailed lower-level claims. For example, to develop a claim of subsystem reliability into sub-claims about the reliability of the constituent components, we can use a strategy of reasoning over *cut-sets*, i.e., the unique combinations of components whose failure leads to sub-system failure. The subsystem architecture serves as associated context, while the subsystem failure analysis provides the requisite justification for using the strategy. Note that we can also perform claims definition and claims refinement by applying, and instantiating, argument patterns.

In a bottom-up development, the strategies chosen must aggregate/compose lower-level claims into higher-level claims. Typically, this is required when solutions (such as the result of a specific verification) are available from which evidence assertions can be inferred, and which must be linked to the higher-level claims that require support. For example, low- level assertions of successful unit testing can be used to (partially) support a software fitness claim, through a strategy of using unit testing, together with the appropriate context and justifications that support the relevance of that strategy.

The result of claims refinement/composition is a collection of fragments of argument structures, whose leaves are claims, evidence assertions, or solutions.

2.3.4 Evidence Definition / Selection

The result of evidence definition/selection is a collection of fragments of argument structures, and corresponding solutions. The former contain claims supported by solutions and are, therefore, useful during a top-down development. The latter mainly comprise solutions with the associated evidence assertions, which can be composed to support higher-level claims (as described earlier in Section 2.3.3).

In a top-down development of safety arguments, we specify evidence requirements in the early phases of system development and safety analysis. As we develop the system, we refine the evidence requirements, and choose/produce those solutions that provide the requisite degree of assurance. For example, a (higher-level) claim of software fitness may be supported by a proof of correctness, as well as through other verification evidence, such as testing, static analysis, etc. In general, the choice of the evidence to be used depends upon on the evidence assertion obtained from claims

refinement, and the degree of assurance required. Thus, a lower-level assertion of functional correctness would be better supported by a proof, whereas an assertion of reliability might necessitate testing of the software, or simulation of a subsystem for a specified duration in an environment with the desired degree of fidelity to actual operations.

When solutions are available, from which evidence assertions can be inferred, then evidence selection also involves determining whether the available solutions are trustworthy, appropriate, and meet the evidence requirements.

2.3.5 Argument Analysis

Argument analysis involves a number of tasks whose goal is to analyze the soundness, or cogency, of the argument structures produced; namely:

A) *Property verification*: We specify (structural) argument properties and verify them to evaluate the *quality* of the argument structure. Properties can reflect concerns such as *well-formedness*, e.g., the argument structure contains no cyclic links, *internal completeness*, e.g., all paths from all claims in the argument structure lead to evidence, as well as more general structural constraints required by best-practices, e.g., a specific type of claim has at least two or more independent paths to, and/or forms of, supporting evidence [54].

B) *Argument validation*: We evaluate the argument structure for possible flaws in the reasoning, i.e., *fallacies* [43], as well as knowledge gaps that lower confidence in claims, i.e., *assurance deficits* [46], and ways to attack an argument, i.e., *defeaters* [81]. Additionally, we *a*) examine the elements of the argument structure for relevance and consistency, and *b*) identify counter-evidence that may either undermine the solutions provided and/or reduce the strength of the argument.

C) *Confidence assessment*: This task supports safety related decision making by characterizing the confidence that can be placed in an argument. The goal is to evaluate the credibility of safety claims on the basis of the *strength* of the argument and the veracity of the evidence supplied. Confidence can be evaluated in a number of ways, e.g., by using probabilistic models, such as Bayesian networks, to specify a joint distribution over the underlying sources of argument uncertainty [25], through belief combination using Dempster-Shafer theory [3], or qualitatively using *confidence arguments* [46].

D) *Metrics-based assessment*: Metrics computed on argument structures, such as size, or coverage [26], are useful to gauge the progress of the safety argument through its evolution. Specifically, metrics provide a convenient summary of the state of the argument development process and thereby a means to gauge whether synchronization with the remainder of the processes in the safety assurance methodology (Fig. 2) is feasible at a given milestone. For instance, to synchronize an architectural safety case (see Fig. 1) with, say, a milestone of design review, we can compute metrics that measure the extent to which higher-level safety claims have been developed into claims concerning the system components and their organization.

2.3.6 Argument Improvement

Whereas argument analysis (Section 2.3.5) is a retrospective activity, argument improvement uses the results of the argument analysis task for (proactively) improving the argument, by: *a*) including the counter-evidence identified, *b*) resolving the identified argument defeaters, *c*) reducing the identified assurance deficits, *d*) modifying the argument structure to address fallacies, and *e*) fixing property violations, if any. Note that some of the argument improvement tasks are not disjoint, e.g., the activities a), b) and c) above. However, they address different (although related) concerns for improving an assurance argument. For instance, assurance deficits broadly concern knowledge gaps in an argument (which may, but need not, include defeaters), whereas argument defeaters are primarily concerned with the sources of doubt.

No specific order is imposed on the tasks, each of which represents a source of change to the argument structure. Thus, improving the argument amounts to determining the exact changes to be made and their scope, e.g., as proposed in [35, 51]. In general, the changes represent editing the argument by adding, removing, and replacing argument fragments. The tasks *a*) and*b*), in particular, may require defining notational extensions to GSN, e.g., as in [40, 46, 51].

3 Requirements for Automation

3.1 Motivating the Need for Automation

Based upon the argument development process (Section 2.3), our experiences in creating real safety cases [6, 13, 15, 24], as well as from feedback from other users and projects within NASA, we have identified a core set of needs, which we believe can be met by providing tool support along with the appropriate automation. Amongst the stakeholders who would benefit from such a capability are safety engineers, systems engineers and developers, managers who are required to make safety-relevant decisions (such as those concerning risk acceptance, effort allocation towards risk reduction measures, etc.) and engineers functioning in an assessment capacity, e.g., regulators, and independent safety auditors.

3.1.1 Maintaining Consistency and Supporting Evolution

As shown in Fig. 1, safety cases evolve during system development and operation. In practice, the processes of system development, safety analysis and argument development are loosely coupled, and consistency between the various artifacts produced is achieved mainly when the processes synchronize, e.g., at defined milestones (Section 2.2).

Since safety cases are meant to be an integrated approach for communicating—to the various participants in system development—the safety concerns, their context, and how those concerns are being addressed, there is a need *i*) to aggregate diverse reasoning and evidence, as well as *ii*) to keep the safety case consistent with the system being developed. During system operation, changes to the system, assumptions

that are validated or invalidated, and observations of safety performance, for example, should translate into updates of the safety case so that the system and its safety case continue to be mutually consistent. Thus, when redesign/replacement of a component is required, we also need to identify those argument fragments that ought to be updated to reflect a necessary, and revised safety analysis, and, in turn, to understand the impact of those changes on the overall safety argument [16].

3.1.2 Structuring Large Arguments

In practice, system safety cases can become substantially large, both as they are being developed, and as they evolve. For instance, the size of the preliminary safety case for airport surface surveillance operations is about 200 pages [34]. Typically, an argument will only be constructed for key parts of the case, with the remainder consisting relevant contextual information, e.g., a hazard analysis and risk assessment. However, depending upon the level of detail required, the underlying arguments also can be large. For example, an end-to-end *slice* of an intermediate safety case of an Unmanned Aircraft System (UAS), which includes software-related claims and formally verification evidence, contains over 500 elements [22]. Effectively, there is a need to support the management of large arguments containing diverse evidence.

Although modular (and hierarchical) organization of arguments can assist in abstracting and reducing the size of a safety argument, thereby potentially improving its understandability, introducing modularity in large arguments without adequate tool support presents additional challenges, including increased effort [52]. This suggests a need for tool support during safety argument abstraction and organization, such that we can create abstract arguments that preserve the reasoning underlying their concrete instances.

3.1.3 Aiding Stakeholder Comprehension

A number of activities need to be performed during the development and management of safety arguments (see Section 2.3). Often, a diverse group of stakeholders, such as systems, safety, and software engineers, is involved. Not all arguments concerning the entire system may be created at the same time, or by the same group of stakeholders. Furthermore, as an argument is created and refined, understanding the relationship between the lower levels of the argument and the higher-level safety claims can be difficult. Each stakeholder, thus, requires the presentation of specific information at a specific point during development, to understand what is being done for safety assurance.

For example, safety case developers may need to determine the claims that remain to be supported, how/if high-risk hazards have been addressed, how a formal method was applied to develop a claim, how a specific pattern has been instantiated, etc. Additionally, since a safety case is one of the primary references for up-to-date safety information it must be understood by the system developers and its operators. In general, there is a need to present role-specific information to subject-matter experts to improve the comprehensibility of a safety argument [15]. Eventually, before a system can be deployed into operation, its safety case must be understood before it

can be subject to analysis and review, e.g., by regulators, and safety auditors. Thus, in addition to supporting stakeholder comprehension of arguments, there is a related need to support analysis and review, which we describe next.

3.1.4 Supporting Analysis and Review

As a system is being developed, engineering artifacts and the safety case must be assessed at different milestones to determine their progress [17]. Both the artifacts and the safety case must be accepted to have met the minimum requirements of a milestone, before development can proceed further. For example, during a Preliminary Design Review (PDR), the architectural safety case can be examined to establish whether or not all the identified safety requirements have been allocated to the relevant system components.

As such, a system safety case can serve as a basis for safety certification. It also provides a framework through which compliance to the relevant regulations, and the corresponding certification requirements can be shown. For example, demonstrating traceability is a major requirement during the certification of aviation systems, as part of the software approval process [75]. An important form of traceability is to show how requirements from regulations, standards, and other relevant guidance documents are linked to the appropriate evidence items. In addition to providing descriptive text—as is the case in practice—the evidence for compliance can include an appropriate slice or *view* of the assurance argument structure, showing the relevant claims, how they have been refined, and eventually substantiated. Based upon the nature of the regulations, existing assurance processes, and the application domain, safety cases may need to be presented with specific content in a particular format, e.g., in the form of reports [48, 73], beyond a presentation of the underlying argument structures. Indeed, some guidelines do not explicitly require an argument [76, 77]. Among those that do, the use of a graphical notation may be an additional requirement [33], or it may be optional [71].

In general, there is a need to support both safety case developers and assessors with argument analysis and review, in particular to extract, present and analyze the required information. As observed earlier, this need is closely tied to the need to aid stakeholder comprehension (Section 3.1.3).

3.2 Requirements Specification

Towards addressing the general needs for safety case development identified earlier (Section 3.1), and the more specific needs to support our process (Section 2.3), we have defined a number of high-level requirements[13] to implement automation support in AdvoCATE (Fig. 4).

Our vision of an automation support solution draws on ideas in model-based development. For instance, argument patterns can be considered as a model from which to generate (instance) argument structures. By design, a pattern is typically smaller

[13] We have also identified and specified additional requirements that cover the remainder of the functionality offered by AdvoCATE, although those are out of scope here.

R1. AdvoCATE shall support automated creation and assembly of fragments of safety arguments.
  i. A mechanism shall be provided to specify argument patterns
  ii. AdvoCATE shall provide functionality to automatically instantiate argument patterns to create instance arguments.
  iii. Formal methods, in particular for software, shall be integrated into safety cases, i.e., there shall be a mechanism to include the evidence produced from, and the reasoning underlying, formal methods for software verification.

R2. There shall be features for complexity management of argument structures.
  i. hierarchization of argument structures shall be supported.
  ii. A modular organization of argument structures shall be supported.

R3. A framework for improved comprehension of safety arguments shall be provided.
  i. Stakeholder-specific querying and viewing of safety arguments shall be supported.
  ii. Safety arguments shall be queried using a query language.
  iii. The results of queries shall be presented as views using a view mechanism.

R4. Evaluation processes for safety arguments shall be supported.
  i. There shall be a capability to compute metrics on safety arguments.
  ii. Customized metrics shall be specified using a metrics specification language.
  iii. A verification language shall be used to specify properties of argument structures.
  iv. A verification environment shall be provided to verify properties of argument structures.
  v. Reports shall be generated from safety argument structures.

Fig. 4. High-level requirements for automation support in AdvoCATE

in size in comparison to its instance. Consequently, we anticipate that comparatively lesser effort could be spent in creating, understanding and evaluating the pattern as opposed to its instance. We believe that one way to address the need to support safety argument evolution, and its consistency with the system being developed/operated (Section 3.1.1), is through a tighter coupling of the different processes, e.g., by automatically transforming system development artifacts, along with the rationale for the methods/tools used, into the argument that the system developed will be fit for purpose (Fig. 4, requirement R1).

For example, in software-intensive systems, a reasonably large number of software requirements can be derived from system safety goals, and the evidence to support those goals can be generated from a variety of formal methods/verification tools. By automating the transformation of the formal verification results or, more generally, by integrating the formal methods used into the argument, we can reflect changes to the requirements, the implementation, and the consequent verification evidence, by regenerating the assurance arguments. Here, patterns serve as the mechanism to encode the reasoning underlying development processes, methods, tools, and the corresponding artifacts produced. Consequently, automated pattern instantiation is a central component of our automation solution.

Subsequently, implementing mechanisms that enable modular and hierarchical organization (Fig. 4, requirement R2) can address the need to structure the arguments being created (Section 3.1.2).

The potential to query a safety case has been previously suggested as one of the benefits of using safety cases, and as a way for stakeholders to understand how safety

concerns have been addressed [10]. Thus, a principled way to specify various kinds of *queries* on the safety argument, e.g., concerning structural properties, traces between artifacts, and domain-specific information, and to generate relevant *views* [18] is another component of our automation solution (Fig. 4, requirement R3). In order for queries to access the relevant information, we augment the safety argument with *metadata*.

Generating reports from the argument structure and its various views, e.g., in a mandated report format, can additionally contribute to assisting how stakeholders understand the safety case. We can, then, extend queries with more expressive logic to provide a language and environment for argument property verification, as well as for computing user-defined metrics. These capabilities (Fig. 4, requirement R4) collectively comprise our approach to respond to the needs of supporting argument evolution (Section 3.1.1), aiding stakeholder comprehension (Section 3.1.3), and supporting argument analysis and review (Section 3.1.4).

4 Foundations

In this section we present the formal foundations of the structures manipulated by AdvoCATE, which we have formalized as various kinds of labeled graph. We present argument structures, briefly explain argument patterns, and describe the hierarchical and modular extensions to arguments. To illustrate the intuition underlying the formalization, we also present the associated GSN for each kind of structure.

4.1 Argument Structures

A (non-modular) argument structure in GSN (Fig. 5) contains a top-level (root) *goal* stating a safety claim. We develop goals into sub-goals using *strategies*, and continue goal development until there are elementary claims that can be connected to the available evidence, i.e., *solutions*. The structure also specifies the *assumptions* made, the *justifications* if any, e.g., for the strategies used or the sub-claims created, as well as the *context* in which the claims, strategies, and solutions are valid.

We link goals, strategies, and solutions using the *Is Supported By* link ($\rightarrow$) while context, assumption, and justification elements require an *In Context Of* link ($\rightarrow\!\!\triangleright$). GSN provides a graphical annotation ($\diamond$) for goals and strategies to indicate that they are *to be developed* (*tbd*), i.e., they are incomplete.

Fig. 5 gives a simple illustrative example. Here, the top-level claim, "G1: Failures of the LiPo battery system are acceptably tolerated", which is made in the context of the failure modes and effects analysis (FMEA) of the LiPo battery system (context node C1) is decomposed by two strategies, S1 and S2, which provide complementary arguments—i.e., over the identified failure modes, and of redundancy, respectively. The latter relies on an assumption of independence in failures of the redundant batteries (assumption node A1), but has not been further developed. The use of the former has been justified (in justification node J1), and results in two sub-goals: G2 (concerning the elimination of short circuits in the battery system), and G3 (concerning

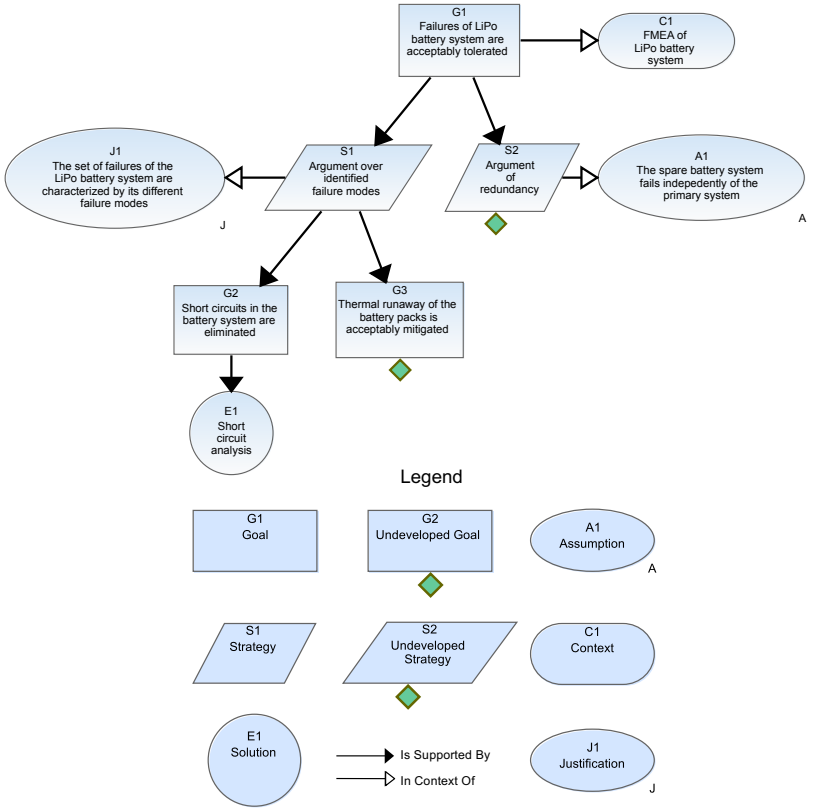


Fig. 5. Basic (non-modular) GSN and a simple example to illustrate usage

the acceptable mitigation of thermal runaway), respectively. The latter remains to be developed, while the former is addressed by evidence node E1, i.e., short circuit analysis.

Note that GSN nodes are intended to be pointers to more detailed information, with the description of the node summarizing those details. Thus, we can give detailed definitions/content externally, and link those to an appropriate node whose description could be, simply, an identifier. For example, the context node C1 of Fig. 5 contains a simple clarifying description, although the content to which it would be linked could be the detailed FMEA report.

Definition 1 (Argument Structure) An *argument structure*, S, is a tuple $\langle N, l, \rightarrow \rangle$ comprising: a set of nodes, N; a family of labeling functions, l_X, where $X \in \{t, d, m, s\}$, giving the node fields *type*, *description*, *metadata*, and *status*; and $\rightarrow$ is the connector relation between nodes. Let $\{s, g, e, a, j, c\}$ be the node types *strategy, goal, evidence, assumption, justification*, and *context* respectively. Then, $l_t : N \rightarrow \{s, g, e, a, j, c\}$ gives

node types, $l_d : N \to string$ gives node descriptions, $l_m : N \to \mathrm{P}(A)$ gives node instance attributes (see below), and $l_s : N \to \mathrm{P}(\{tbd\})$ gives node development status[14]. We require the connector relation to form a *finite forest*, with the operation $isroot(\to, r)$ checking if the node r is a root of the forest[15].

Furthermore, the following structural conditions must be met:

1) Each root of the argument structure is a goal: $isroot(\to, r) \Rightarrow l_t(r) = g$
2) Connectors only leave strategies or goals: $n \to m \Rightarrow l_t(n) \in \{s, g\}$
3) Goals cannot connect to other goals[16]: $(n \to m) \wedge [l_t(n) = g] \Rightarrow l_t(m) \in \{s, e, a, j, c\}$
4) Strategies cannot connect to other strategies or evidence: $(n \to m) \wedge [l_t(n) = s] \Rightarrow l_t(m) \in \{g, a, j, c\}$
5) Only goals and strategies can be undeveloped: $tbd \in l_s(n) \Rightarrow l_t(n) \in \{g, s\}$.

Note that Definition 1 does not need to distinguish *is supported by* edges ($\to$) that connect core nodes, and *in context of* edges ($\to\!\!\triangleright$) that connect contextual nodes. Instead, we make this distinction with a notational convention, writing $v_1 \to\!\!\triangleright\ v_2$, if $v_1 \to v_2$ and $l_t(v_2) \in \{a, j, c\}$.

We have additionally extended GSN nodes with *metadata*, which we associate with individual nodes (rather than globally with the entire argument or pattern) in the form of a set of associated instance attributes. Attribute types are declared globally and can be parameterized over parameters of specific types. Nodes have instances of attributes with values that comply with the type of the parameter (which can itself depend on the node).

The type of a parameter can either be:

- a basic type, i.e., a string (String), an integer (Int), or a natural number (Nat)
- a *node type*, which can be used as parameters in three different ways:
  - NodeID: any kind of node
  - sameNodeTypeID: the parameter must be the identifier of a node of the same type as the node with the attribute.
  - Specific node parameter types, which allow specification of a node of a given type: assumptionNodeID, contextNodeID, evidenceNodeID, goalNodeID, justificationNodeID, strategyNodeID.
- A user-defined enumeration (userDefinedEnum): for example, we can define the parameter types

  severity ::= catastrophic | hazardous | major | minor | noSafetyEffect
  likelihood ::= frequent | probable | remote | extremelyRemote | extremelyImprobable

  to define the parametrized attribute risk(severity, likelihood). Then, we can give an *attribute instance* as: risk(severity(catastrophic), likelihood(extremelyImprobable)). We will just use 'attribute' when it is clear from the context whether

[14] Status is defined as a set since, as we will see later, nodes can have multiple status values. Here, *tbd* represents the '*to be developed*' status.

[15] A partial argument can have multiple roots, whereas a *full* argument structure has a single root.

[16] Formally, we define a strict notion of argument where goals require intermediate strategies (thus spelling out explicitly why subgoals follow from parent goals), and separate goals cannot share evidence. In practice, both these conditions are often violated and can be captured with a more relaxed definition. The tool allows both conventions.

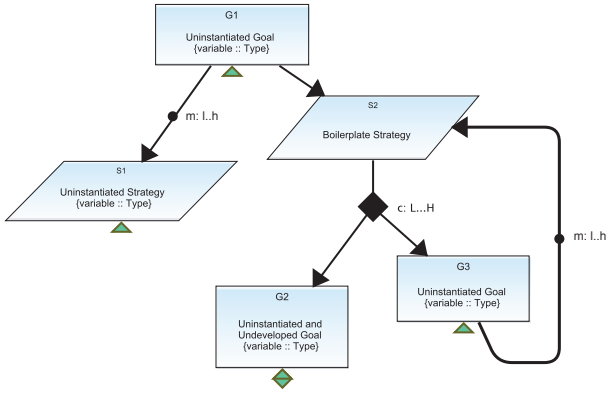


Fig. 6. GSN for argument patterns.

we mean attribute instance or attribute declaration. Note that we do not force the values of different enumerations to be distinct.

Metadata can be used to express relations between nodes, relations between argument nodes and patterns (indicating how instance nodes have been generated; see Section 4.2), tracing information (e.g., to external artifacts, such as goals to requirements, or to risks and hazards), provenance information for the integration of external sources, such as formal methods. For example, to reflect the notion that a particular node in a pattern formalizes another node of the same type in that pattern, we can use the attribute: `formalizes` (sameNodeTypeID).

4.2 Argument Patterns

Safety argument patterns [20, 23] capture repeatedly used structures of successful, i.e., correct, comprehensive and convincing arguments, within a safety case. In effect, they provide a re-usable approach to safety argumentation by serving as a means to capture expertise, so-called *tricks of the trade*, i.e., known best practices, successful certification approaches, and solutions that have evolved over time.

The existing notion of a safety argument pattern is an extended argument structure, often specified graphically using GSN, which abstractly captures the reasoning linking certain (types of) claims to the available (types of) evidence, and is accompanied by a clear prescription and proscription of its usage. Fig. 6 shows the GSN used to specify argument patterns along with notational extensions that we defined in our earlier work [20]. There are two types of abstractions for pattern specification:

i) *Structural abstraction* applies to the $\rightarrow$ and $\multimap$ relations, providing the concepts of *multiplicity* and *choice*. The former abstracts over repeated elements, whereas the latter captures a choice between alternatives (which may or may not be exclusive). As shown in Fig. 6, we indicate multiplicity by placing a '$\bullet$' on the

relevant link accompanied with an annotation, $l \ldots h$, indicating its lower and upper bounds. When the bounds are $0 \ldots 1$ the annotation represents *optionality* and is indicated by placing a '$\circ$' on the link instead. Choices are shown using a '$\blacklozenge$' between the links, also annotated with the bounds of the choice, $L \ldots H$.[17]

ii) *Entity abstraction* in GSN provides the concepts of *uninstantiated* (or *to be instantiated*), and *uninstantiated and undeveloped* (i.e., *to be instantiated* and *to be developed*) elements. We represent the former by annotating the corresponding nodes with '$\triangle$', referring to abstract elements whose parameters are replaced with concrete values upon instantiation. We represent the latter by annotating the relevant nodes with a combination of '$\triangle$' and '$\diamond$' (i.e.,' $\hat{\diamond}$ '; see node G2 in Fig. 6), and it indicates uninstantiated entities that are *also* to be developed. Thus, upon instantiation, an abstract uninstantiated and undeveloped entity is replaced with a concrete, instantiated, but still undeveloped, instance. Patterns thus abstract over possibly undeveloped, but instantiated, fragments.

We specify the parameters to be instantiated as $\{v :: T\}$, where v is the parameter variable and T is its type. Additionally, although not part of the GSN standard, there are a few examples of the use of a *recursion* abstraction in the literature [58], which we have also formalized. Fig. 6, shows recursion using the loop link between the goal node G3 and strategy node S2, to expresses the idea that a pattern (or a part of it) can be, itself, repeated and unrolled, e.g., as part of an optional link, or a larger pattern [20]. Note that although loops are permitted in patterns, they are prohibited in arguments. Also see Fig. 15 for an illustrative example of patterns. In particular, Fig. 15b gives an example of a pattern using the recursion abstraction.

Definition 2 (Argument Pattern) An *argument pattern* (or *pattern*, for short), P, is a tuple $\langle N, l, p, m, c, \rightarrow \rangle$, where: $\langle N, \rightarrow \rangle$ is a directed hypergraph in which each hyperedge has a single source and possibly multiple targets; l is a family of labeling functions, l_X, where $X \in \{t, d, m, s\}$; and p, m, and c are additional labeling functions. The structural conditions from Definition 1 hold, as well as the following conditions below:

1) l_X, where $X \in \{t, d, m\}$ is as in Definition 1. We have $l_s : N \rightarrow \mathrm{P}(\{tbd, tbi\})$, retaining the status restriction from Definition 1 (item 5).

2) p is a parameter label on nodes, $p : N \rightharpoonup Id \times T$, giving the parameter identifier and type. Without loss of generality, we assume that nodes have at most a single parameter

3) $m : N^2 \rightharpoonup \mathbb{N}^2$ gives the multiplicity range on a link between two nodes, where $m(x, y)$ is defined iff we have $x \rightarrow Y$ and $y \in Y$. If Y has more than one node, i.e., y is part of a choice, we assume, without loss of generality, that the multiplicity applies to the outgoing connector to y. Multiplicity bounds of $\langle L, H \rangle$ represent the range $L..H$, where $L \leq H$ and $0 < H$. An optional connector has range 0..1.

4) $c : N \times \mathrm{P}(N) \rightharpoonup \mathbb{N}^2$, gives a "$L..H$ of n" range on the choice attached to a given node, where $c(x, Y)$ is the choice between Y with parent node x, where $c(x, Y)$ is

[17] Though not so common in practice, bounds are a natural generalization of optionality and multiplicity and can be used, for example, to require complementary strategies or evidence (as a lower bound on number of branches) or to limit argument complexity (by placing an upper bound).

defined iff we have $x \to Y$. Here, n is simply the number of legs in the choice, and so can be omitted. We require the bounds on choices to be within the number of legs of the choice, i.e., if $a \to \{b_1, \cdots, b_n\} = \mathbf{B}$ and $c(a, \mathbf{B}) = \langle L, H \rangle$ then $0 < L \leq H \leq n$ and $L < n$.

There are additional restrictions on argument patterns, namely: *a*) the *multiplicity condition*, i.e., multiplicity on a link in a pattern that is followed by a *boilerplate* node (a node without a parameter) must eventually be followed by a *data* node (a node with a parameter); *b*) the *single parent condition*, i.e., patterns can contain nodes with multiple parents, but their instances cannot; and *c*) *well-foundedness*, i.e., patterns containing loops cannot contain inescapable paths. See [20] for more details.

4.3 Hierarchical Arguments

We can introduce hierarchical structuring into *flat* argument structures (Fig. 7). A *hierarchical node* (or hinode) groups a fragment of an argument structure into an abstract node. Hierarchical nodes can be *open*, so that the argument structure that they contain is visible, or can be *closed*, and can be viewed simply as a (hierarchical) safety case node.[18] There are conditions on which fragments can be abstracted [27].

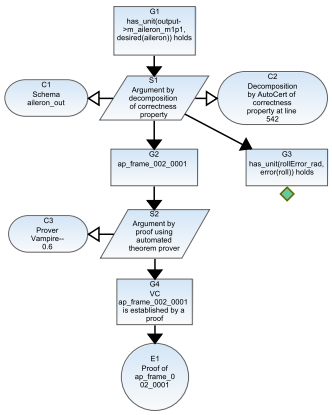
We need the following notions. A fragment is *fully developed* if all goals lead to evidence and *well-developed* if the root is a goal and the leaves are goals or evidence.

There are three types of hinodes:

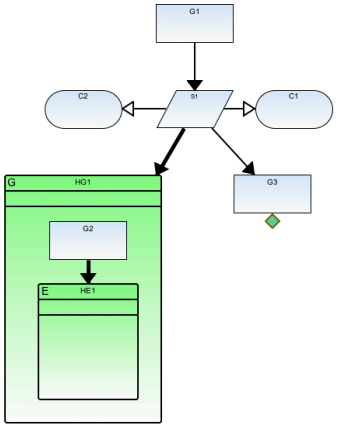
1) *Hierarchical goals* (Fig. 7b; *higoals*, for short) abstract *well-developed* argument fragments (for example branches which end in goals or evidence); one of their main purposes is to provide a high-level view of an argument structure.
2) *Hierarchical strategies* (Fig. 7c) represent the decomposition of a goal to several subgoals by aggregating a chain of strategies (with intermediate goals), and hiding parts of the argument that are considered supplemental to the main argument of interest.
3) *Hierarchical evidence* is the special case of a hierarchical strategy with no outgoing goals, but where the subtree encloses a *fully developed* fragment (e.g., a proof represented as argument fragment). In Fig. 7d, since the fragment starting from the strategy S2 is downwards complete (has no undeveloped elements), we can construct a hierarchical entity that abstracts and encapsulates it. In other words, the subtree that justifies goal G2 can be packaged as the hierarchical evidence node HE1. The hinode HE1 in Fig. 7d, is in its open view, where both the hinode and the structure it abstracts are visible. Similarly, in Fig. 7c, the hierarchical strategy HS1 is in its open view, whereas the included hierarchical evidence HE1 is in its closed view.

We now extend Definition 1 to *hierarchical argument structures* (also called *hicases* for short). Hicases extend the definition of safety cases with an additional relation to represent hierarchical structure. We use the partial order symbol $\leq$ where

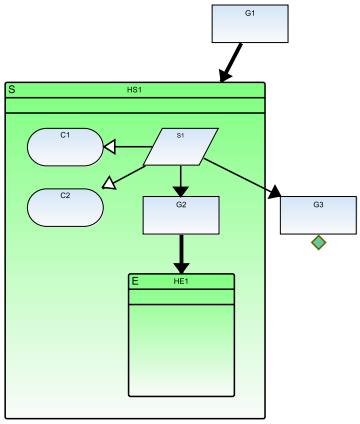
[18] Note that the *open* and *closed* views of a hinode, respectively, serve to visually display or hide the node contents. In our current implementation, a hinode cannot be empty regardless of whether it is displayed in its open or closed view, although as part of future work we plan to allow the creation of empty hinodes.



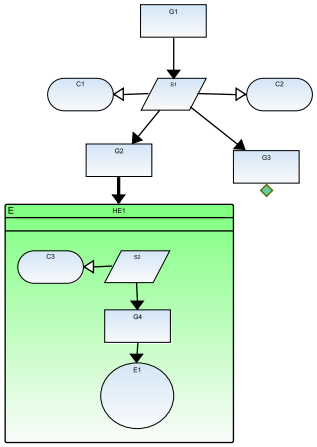
(a) Flat argument structure fragment



(b) Hierarchical goal (open view); hierarchical evidence (closed view)



(c) Hierarchical strategy (open view); hierarchical evidence (closed view)



(d) Hierarchical evidence (open view)

Fig. 7. *Flat* argument structures and hierarchization. Note that the description of the nodes has been omitted in the hicases shown in (b), (c), and (d), since the objective here is to illustrate the different hinodes.

$n < n'$ means that the node n *is inside* n'. We wish to define hicases in such a way that we can always *unfold* all the hierarchy to regain a flat safety case.

Definition 3 (Hierarchical Argument Structures) A *hierarchical argument structure* (or hicase, for short) is a tuple $\langle N, l, \rightarrow, \leq \rangle$. The set of nodes N, labelling function l, and connector relation $\rightarrow$ are as given in Definition 1. The forest $\langle N, \rightarrow \rangle$ is subject to all conditions except condition (1) of Definition 1 which is generalized to state that global roots must be goals: $isroot(\rightarrow, v) \wedge isroot(\leq, v) \Rightarrow l_t(v) = g$. The hierarchical

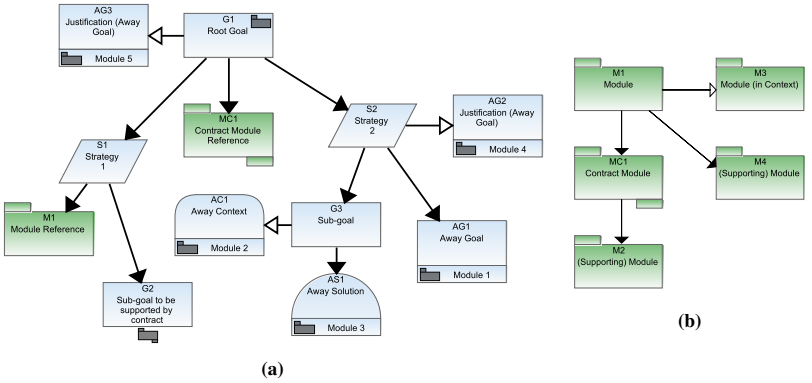


Fig. 8. GSN extensions for modular organization: (a) Intra-module GSN; (b) Inter-module GSN.

relation is a (necessarily finite) forest *qua* poset $\langle N, \leq \rangle$. Finally, we impose several conditions on the interaction between the two relations $\rightarrow$ and $\leq$ (omitted here; see [27] for details)

The conditions allow us to derive the type of a hinode from its contents (and surrounding context). Hinodes also have attributes in the same manner as their underlying counterparts; however, as for node types, some of the attributes for hinodes must be consistent with those of the nodes they enclose. For example, a hinode is considered undeveloped if any of its contained nodes is undeveloped.

4.4 Modular Arguments

In the GSN, there are *intra-module* and *inter-module* notational extensions for presenting modular arguments (Fig. 8). In addition to modules, GSN also provides a concept of *contract module*, a specialized module containing a definition/justification of the relationship between two or more modules, in particular how a claim in one (or more) module(s) support(s) the argument in the other(s) [39]. As originally conceived, (argument) contracts were intended to represent relations between goals, context and evidence of modules participating in a composition, and were documented using *contract tables*. In order to integrate a contract and view it within the same framework as the argument, rather than as an external entity, the use of GSN itself to specify contracts within contract modules, has been proposed [36], and this is the convention we will follow here.

Intra-module GSN is used within a specific module to reference other modules, using *module reference* nodes, and specific elements in other modules, using so-called *away* nodes. An *away goal* in one module repeats a claim present in another module. Similarly, an *away context* and an *away solution* in a module repeat, respectively, the context and a reference to evidence items present in another module. Each away node additionally contains a reference to the module containing the original content.

For example, in Fig. 8a the away goal AG1 repeats the claim present in some goal node in the module Module 1. Away nodes in the local module are *public* nodes in the referenced module and are indicated by the '🗀' annotation placed at the top right of the node (as shown for the goal node G1 in Fig. 8a). Thus, a public node of a given type, i.e., a goal, context, or evidence, is visible to other modules, and can be referenced within those modules by using an away node of the corresponding type.

When the argument is supported in an, as yet, unspecified module but the contract of support is available in a contract module, the reference to that contract is shown using a *contract module reference* node (e.g., node MC1 in Fig. 8a). When the contract itself is unspecified, a *to be supported by contract* annotation (🗏) is used (e.g., as shown for goal node G2 in Fig. 8). This notation is analogous to, and mutually exclusive with, the *to be developed* annotation in non-modular GSN (Fig. 5).

As shown in Fig. 8a, intra-module GSN permits links to away goals using both the $\rightarrow$ and $\multimap$ relations. The former implies claim refinement, i.e., by a claim in a different module as indicated by the away goal, whereas the latter is a substitution for a justification node, but where (a) *more* justification is required than can be provided by a justification node alone, and (b) where the additional justification is provided in a different module.

The use of inter-module GSN effectively specifies a *module view* that is intended to show modules and their relationships (Fig. 8b). As shown, modules can be linked to: *a*) other modules, by either the $\rightarrow$ or $\multimap$ relations. For example, in Fig. 8b, M1 $\rightarrow$ M4 means that there exist one or more goal and/or evidence nodes in module M4, that support, respectively, one or more strategies or goals in module M1. Additionally, M1 $\multimap$ M3 means that there are one or more goals or strategies in module M1, that have a contextual reference to one or more context and/or goal nodes in module M3; and *b*) contract modules, using the $\rightarrow$ relation: the contract module explicates the support relationship between the modules to which it is linked.

Modularity allows safety cases, as with other artifacts, to be decomposed into discrete modules so as to contain change impact, support distributed development, etc. Hierarchy, on the other hand, permits a system to contain sub-systems of the same kind. The concepts are thus distinct, though complementary; if modules can themselves contain modules, this results in hierarchical modularity [11].

We now formalize the notion of a single modular argument structure, i.e., an individual diagram, after which we extend the formalization to inter-connected collections of modules.

Definition 4 (Modular Argument Structure) Let $\{s,g,e,a,j,c\}$ be the node types as in Definition 1. We extend these with two additional node types: *mr*, and *cr*, denoting *module reference* and *contract module reference* respectively. A modular argument structure, (or module, for short), M, is a tuple $\langle N,l,t,d,\rightarrow\rangle$, where

- N and $\rightarrow$ are as in Definition 1;
- d is a module description string;
- l is the same family of functions as in Definition 1, where
  - $l_t : N \rightarrow \{s,g,e,a,j,c,mr,cr\}$ gives node types
  - $l_d : N \rightarrow string$ gives node descriptions;
  - $l_m : N \rightarrow \mathrm{P}(A)$ gives node instance attributes, and

  - $l_s : N \to \mathrm{P}(\{tbd, tbsbc, public, away, contextual\})$, gives node status, i.e., whether a node is, respectively, *to be developed*, *to be supported by contract*, declared *public*, references an *away node*, or is *used in context*.
- t is a family of functions that give the target of nodes that reference other modules. Let I_m and I_n be sets of identifiers distinct from N, representing modules (and contracts) and nodes external to M, respectively. Then, for module reference, x, $t_r(x)$ gives the target module, and for an away node, $t_a(x)$ gives the pair of module and public node, i.e., $t_a : \{n \in N \mid away \in l_s(n)\} \to \mathrm{I}_m \times \mathrm{I}_n$, and $t_r : \{n \in N \mid l_t(n) \in \{mr, cr\}\} \to \mathrm{I}_m$.

Additionally, we require individual modular argument structures to form forests, and various structural conditions (omitted here) to hold.

Contracts can be defined as a particular kind of modular argument[19] and are used to abstract the relation between producer and consumer modules.

A set of interrelated modules and contracts forms a hierarchy, subject to various constraints. Assume a set of IDs for modular argument structures (I_a), contractual argument structures (I_c), module containers (I_m), and mappings (M_a, M_c) to the sets of modular arguments (**A**), and contracts (**C**), respectively.

Definition 5 (Module Hierarchy) A *module hierarchy*, H, is a tuple $\langle \mathrm{I}_m, \mathrm{I}_a, \mathrm{I}_c, \mathbf{A}, \mathbf{C}, \mathrm{M}_a, \mathrm{M}_c, < \rangle$, comprising distinct sets of

- module container IDs, I_m;
- modular argument IDs, I_a;
- contractual argument IDs, I_c;
- modular arguments, **A**;
- contractual arguments, **C**; and
- mappings $\mathrm{M}_a : \mathrm{I}_a \to \mathbf{A}$, and $\mathrm{M}_c : \mathrm{I}_c \to \mathbf{C}$,

along with a forest $\langle \mathrm{I}, < \rangle$, where $\mathrm{I} = \mathrm{I}_a \cup \mathrm{I}_c \cup \mathrm{I}_m$, such that $i \in \mathrm{I}_a \cup \mathrm{I}_c \Rightarrow leaf(<, i)$, and $root(<, i) \Rightarrow i \in \mathrm{I}_m$.

The forest represents the *containment relation* between modules (which is distinct from the hierarchy relation on arguments; Section 4.3). Since it is a forest, there need be no single top-level module. A module hierarchy represents a snapshot of a possibly incomplete collection of safety arguments under development and, thus, although arguments and patterns must be leaves, we do not require all leaves to be arguments and patterns. That is, during development, we allow a module leaf (with no argument within). We also allow a tree with a single node, i.e., an empty module. Since Definition 5 allows forests, we allow multiple arguments in a single module, and multiple argument fragments (with distinct roots) in a single argument.

Finally, we can define notions of *well-scoping* (references between modules exist are are correctly scoped according to the module hierarchy, and there are no reference cycles) and *well-formedness* (references between modules have the appropriate type and data).

In Section 6.3 we describe tool functionality for creating modular arguments as well as *module views* (i.e., inter-module GSN).

[19] Strictly speaking, they relax some conditions on the definition of modules and add others.

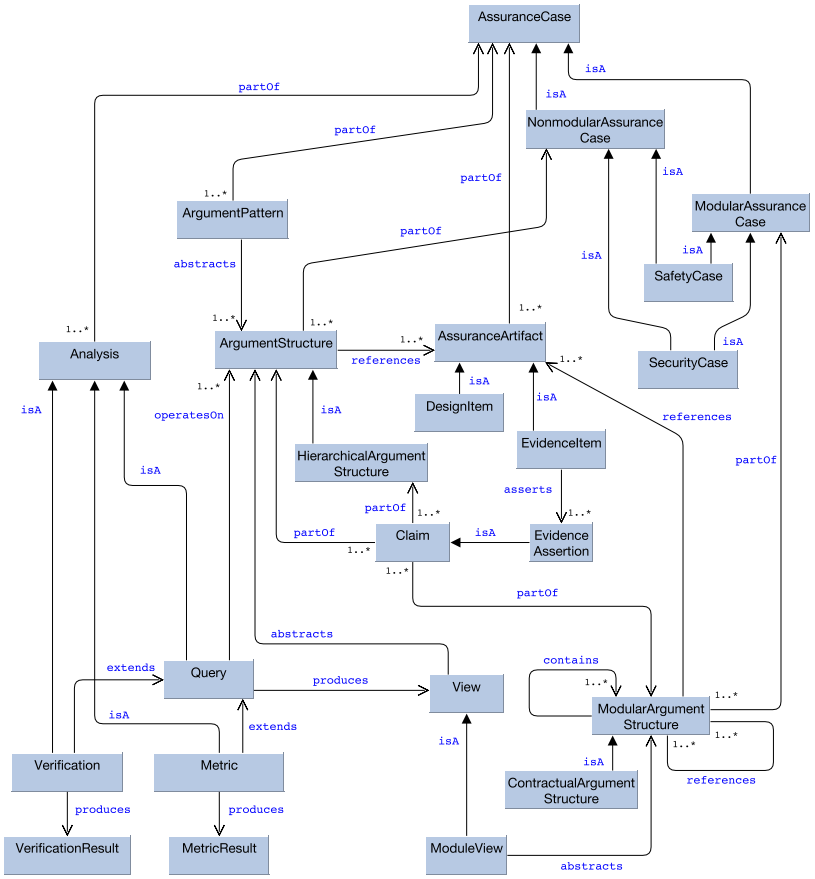


Fig. 9. Conceptual model for argumentation elements and extensions supported by AdvoCATE

5 System Description

5.1 Conceptual Model

The preceding section provided formalizations of the various elements of an assurance case. Before describing the implementation (Section 5.2), we first give an informal (and incomplete) conceptual model (Fig. 9) to indicate some of the relationships between the various high-level concepts. Assurance cases can have a modular or non-modular structure, and can address a range of assurance concerns for a system. In the context of our practice, safety is most commonly the primary concern (for which we produce safety cases), although another related concern is security. Thus, a *security case* is another kind of assurance case.

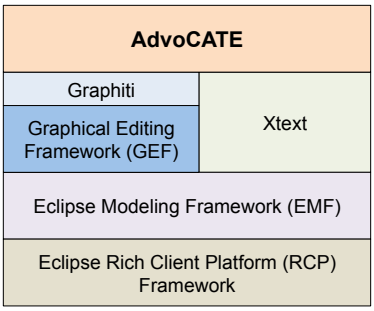


Fig. 10. AdvoCATE execution stack.

Conceptually, assurance cases can be considered to consist of a collection of (modular or non-modular) argument structures, argument patterns, associated assurance artifacts (including evidence items linked to from solution nodes, and possibly other artifacts providing additional context, linked from other nodes), along with analyses (e.g., queries, verifications, and/or metrics). System assurance concerns of interest are captured in a number of claims in the modular or non-modular argument structures. The latter also can be hierarchical, although currently we do not combine hierarchy and modularity. As mentioned earlier (Section 2.3.2), an evidence assertion is a particular kind of claim relating to an evidence item. Views either provide an abstraction of the modular structure, or are generated by applying queries to an argument. Finally, a modular assurance case consists of a collection of modular argument structures which are organized using two relations: inter-module references and the module hierarchy (capturing containment).

5.2 Implementation

We have implemented AdvoCATE using the Eclipse framework[20] as a collection of *plugins*, bundled into independent features, each of which can be separately added or removed. Plugins use the Eclipse rich client platform (RCP) for the user interface, and the Eclipse modeling framework (EMF) for the underlying data model. We use Graphiti[21], which in turn uses the Eclipse graphical editing framework (GEF), to provide the graphical representation of the diagrams, and an editor for manipulating the diagrams. Additionally, we use the Xtext framework[22] to implement (domain-specific) languages for queries, verification, and report generation. The Xtext framework also provides support for creating the language infrastructure, i.e., the editor, parser, linker, etc., as well as usability features in the generated editor, such as syntax highlighting, syntax completion, etc. Fig. 10 shows the AdvoCATE execution stack, highlighting these various elements.

[20] Available at: http://www.eclipse.org/

[21] Available at: http://eclipse.org/graphiti/

[22] Available at: http://eclipse.org/Xtext/

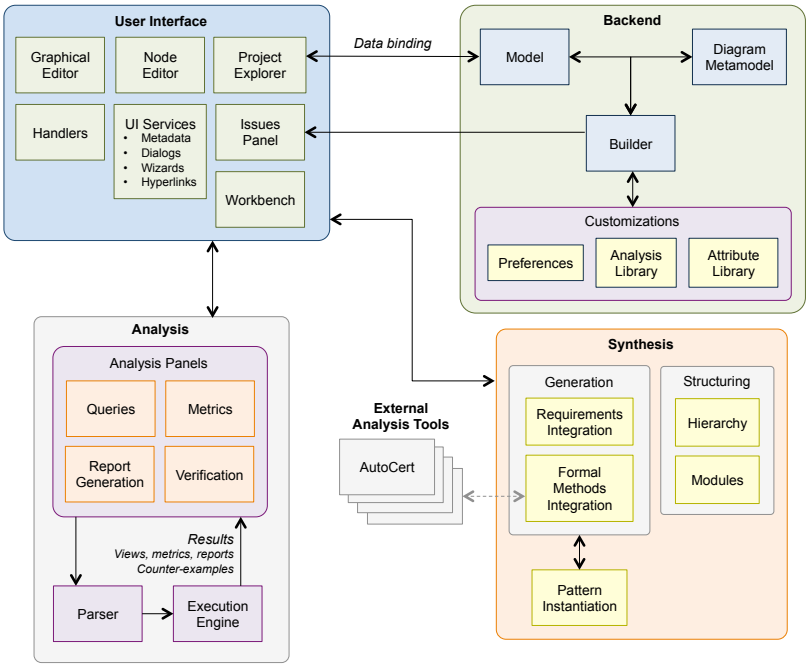


Fig. 11. AdvoCATE system architecture

As shown in Fig. 11, AdvoCATE has been architected around four main subsystems: the user interface (UI), the backend, and various analysis and synthesis functionalities. Note that, to avoid clutter, not all links between (and within) the corresponding components have been shown. If multiple components of a subsystem connect to another subsystem, this is indicated as a single arrow between the subsystems, but if the primary link is between specific components, then a direct arrow between those components is shown. Also not shown is the interface between the backend, the filesystem and the underlying Eclipse infrastructure. We describe each subsystem next.

5.2.1 Backend

The *diagram metamodel* specifies the elements and format of arguments and patterns, including the core GSN, its modular extensions, our extensions for hierarchy, as well as metadata. Argument and pattern models[23] exist both as a collection of file resources—i.e., extensible markup language (XML) files distinguished within

[23] Henceforth, we will use argument (or pattern) when we mean the model of the argument (or pattern), i.e., the instance of the diagram metamodel.

the filesystem through their extensions, `.argument`, and `.pattern` respectively—and via a data binding to pictogram elements (within Graphiti) that form the visual representation of the GSN diagrams.

The data binding maintains consistency between visual elements and the model. Amongst other things, it updates derived metadata and modifies diagrams when required in order to ensure consistency, e.g., when a node status is set to *tbd*, its image must be updated with the appropriate annotation. Similarly, when a parameter is added to a pattern node, its metadata must be updated accordingly.

In addition to created arguments and patterns, user data consists of the *attribute library* (attributes, parameters, synonyms) and the *analysis library* (queries, metrics, etc.), as well as various tool preferences, such as the status of the well-formedness rules. The model must comply with the diagram metamodel by construction. The *builder* provides a collection of monitors that analyze the data (i.e., the models and user-defined metadata) for compliance with the well-formedness rules (Section 6.1), after every save operation. The builder then updates the issues panel in the UI with the results of the analysis.

5.2.2 User Interface

The main elements of the user interface (UI) (Section 6.1) are the graphical editor for constructing and editing the argument, the node editor where details of individual nodes are edited, the project explorer, which lists files in the various projects, including views and metrics associated with given arguments, and the issues panel, where errors and warnings are reported. In addition, there are several other wizards and interfaces developed to support specific fuctionality. Many of these interfaces build on the Eclipse workbench for UI elements. The analysis panel is also part of the UI and is described below. Handlers carry out a number of custom commands invoked from interface elements, such as creation, modification, and deletion of various artifacts (modules and projects) and export of diagrams in different formats.

5.2.3 Analysis and Synthesis

The analysis panels of the tool provide the interface to its functionality to specify and execute queries (Section 6.4), specify and verify properties (Section 6.5), specify and compute metrics (Section 6.6), as well as generate reports.

The analysis component has been implemented using the Xtext framework, which generates functionality from the language grammars, such as parsers for the query, verification and metrics languages. An execution engine takes the parsed input to produce views from queries, compute the metrics, verify or generate counter-examples to a property and create reports.

The *generation* component implements functionality for requirements integration and formal methods integration, both of which invoke the pattern instantiation mechanism. External analysis tools are wrapped as Eclipse plugins, which translates between tool output (XML in the case of AUTOCERT; see Section 6.2.2) and AdvoCATE's internal argument representation. These feed into the formal methods interface allowing their output to be used to create argument fragments.

The *structuring* component allows arguments to be structured by the introduction of well-formed hierarchical nodes; additionally, it provides functionality to create modules, and the related nodes for defining the inter-module links.

6 Functionality

In this section, we present the functionality offered by AdvoCATE. First, we give a brief overview of the basic capabilities of the tool (Section 6.1), after which we describe its automation features. In particular, we describe how we use*i*) pattern instantiation, to automatically create and assemble argument fragments (Section 6.2); *ii*) hierarchization and (manual) modularization, for structural abstraction (Section 6.3); *iii*) queries and views, to present stakeholder perspectives (Section 6.4); and *iv*) specification and automatic verification of argument properties, to support argument analysis (Section 6.5).

6.1 Basic Functionality

Fig. 12 shows a screenshot of the basic interface available to the user whilst creating and editing safety arguments using AdvoCATE. Though our main aim in developing AdvoCATE is to provide automation features during safety argument development, the tool also offers core functionality to support manual development and editing.

For instance, we can manually create (both modular and non-modular) arguments and patterns by placing the relevant nodes and links—provided in a *palette* (top right panel, Fig. 12)—on to a *canvas* (center panel, Fig. 12), or by using pre-defined keyboard shortcuts. When a structure grows large, e.g., when it exceeds the size of the visible canvas, we can create so-called *associated diagrams*, which allow the user to continue creating the argument structure on a new canvas. Multiple canvases can be simultaneously opened in *tabs* for concurrently editing multiple arguments, whereas a *miniature view* (bottom left panel, Fig. 12) provides a bird's eye view of the argument contained on the selected canvas, and also highlights the active area of the canvas being displayed/edited. Node contents, metadata, hyperlinks to external content, and node status can be viewed/edited through a *detail* panel underneath the canvas. In addition to creating and editing the argument, standard features include copying, pasting, and exporting to a variety of formats, e.g., portable document format (PDF).

The tool allows patterns to be copied into arguments as part of a *manual* instantiation process. The user then needs to resolve resolve pattern elements such as parameters and choices. The tool also provides an *automated* instantiation feature, described in Section 6.2. Each time we create and save a diagram, the tool performs a number of pre-defined structural well-formedness checks and displays the results in a number of ways, as shown in Fig. 12:

a) in an *issues* panel, highlighting specific errors and warnings. Here, selecting a specific error/warning provides navigability to the argument element responsible for the issue;

b) on the canvas (if open), highlighting the offending elements (see the goal nodes G11 and G4 in Fig. 12); and

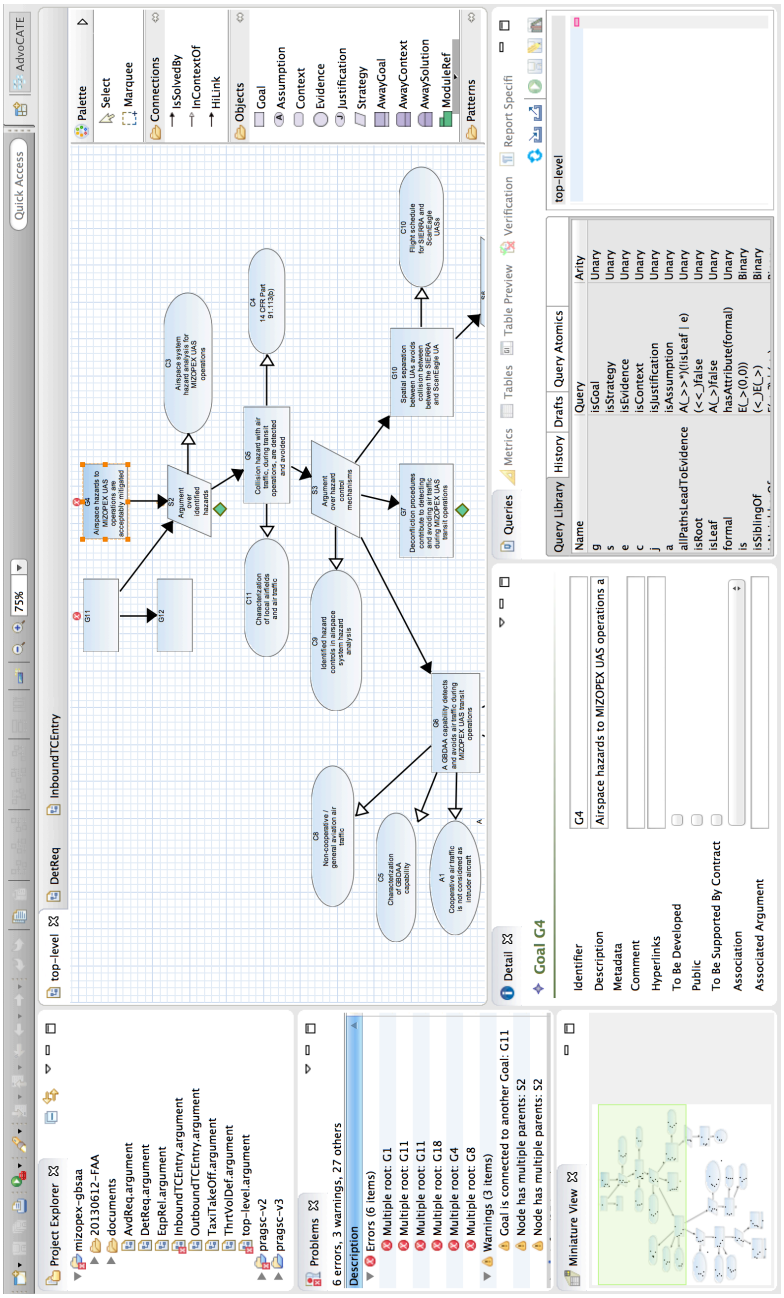


Fig. 12. Screenshot of AdvoCATE interface during safety argument development.

c) in the *project explorer*, highlighting those projects and files containing arguments and patterns that failed the checks. Projects can be considered as a logical collection (of various types of files and folders) that would be recognized by AdvoCATE.

As part of the tool configuration, well-formedness checks can be specified to apply to arguments and patterns, and are user-modifiable to offer three categories of results:

i) *Error*: The structure fails the constraint being checked and ought to be corrected as soon as possible;
ii) *Warning*: The structure fails the constraint but is acceptable as an intermediate state for practical purposes; and
iii) *Ignore*: Omit the check.

The default settings for the checks are such that these categories correspond, roughly, to aspects of the formal definitions. An error reflects an inherently wrong and, therefore, non-well-formed structure, whereas a warning reflects a relaxation of the structural constraints of Definitions 1 and 2. In particular, we check the following concerns:

Cyclicity: More specifically, the lack of cycles in an argument, although cyclic structures are permitted in patterns (See Section 4, Definitions 1 and 2). Thus, the cyclicity check would be configured to produce errors for arguments that have cycles[24], and would be ignored for patterns.

Unique identifiers: Although alphanumeric node identifiers are auto-generated, they can be subsequently modified by the user, e.g., to be more meaningful. This check ensures that, upon user modification, two or more nodes in a given fragment do not have the same identifier, and is configured to produce errors if they do.

Multiple roots: As in Definitions 1 and 2, full argument structures and patterns have a unique root. Thus, this check is normally configured to produce errors, e.g., as shown[25] for the argument fragment in Fig. 12.

Multiple parents: In particular, our formal definitions (Definitions 1 and 2) prohibit multiple parents for any given node, although in practice such structures are not uncommon (e.g., when sharing evidence). This check would be configured to produce warnings for arguments, e.g., as shown in for the argument fragment in Fig. 12, and ignored for patterns.

Intervening strategies between goals: As in Definition 1, we prohibit direct goal-to-goal links and require an intervening strategy. However, once again, such structures are not uncommon in practice and the constraint can be practically relaxed (for both arguments and patterns) by configuring the check to produce warnings.

Consistency of syntax: In particular, for pattern nodes, this check verifies that those nodes with status *tbi* contain a node parameter and an appropriate type (See Definition 2), and produces an error upon failure of the check.

[24] The only way in which cycles can be introduced into arguments is by pasting a pattern with a cycle into an argument.

[25] Here, note that the multiple roots errors shown in the *issues* panel has identified two goal nodes with the same identifier. However, since the check is performed across all the open projects, the errors exist in separate arguments; the path to those arguments can be seen by expanding the size of panel, but has not been shown in Fig. 12.

```
1 Instantiate(P, τ) begin
2 |  I ← { }
3 |  foreach r = (j, v) ∈ τ do
4 |  |  F ← instantiateRow(P, v)
5 |  |  I ← connect(F, j, I)
6 |  end foreach
7 end
```

Fig. 13. Abstract algorithm for pattern instantiation.

In addition to these basic features, additional panels are available to: create and/or modify module hierarchies (see Section 6.3.1, Fig. 21), specify user-defined checks (verification), as well as queries, metrics, and template specifications for report generation (as shown in Fig. 12). Collectively, these represent the AdvoCATE interface for some of the automation features, which we describe next.

6.2 Automated Argument Creation and Assembly

As mentioned earlier (in Section 3.2), a key idea underlying automated creation and assembly of argument structures is specifying and *instantiating* a pattern. In particular, we specify a higher-level abstraction that applies for a given assurance problem—i.e., an argument pattern—so that automated instantiation can address the lower-level details.

Pattern instantiation has been used to auto-generate argument fragments for the safety assurance of both manned and unmanned aircraft systems (UASs): specifically to create safety arguments from a hazard and safety requirements analysis of the Swift UAS [19] and a transport category twin-engine aircraft model [12]; and to integrate formal methods, in particular the results of formal verification [22].

We can instantiate a pattern either interactively, or using data extracted from external sources, e.g., using the output from a requirements management tool, or a formal verification tool. We have implemented mainly the latter in AdvoCATE (illustrated next, in Section 6.2.1), although a restricted form of interactive instantiation can also be invoked to formalize informal claims using a *claim formalization* pattern [23] (which we will illustrate in Section 6.2.2).

In general, the data required to instantiate a pattern can be given as a mapping from (the identifiers of) the data nodes to lists of values that enumerate the possible paths through the pattern. We can conveniently represent this data as a table (τ), whose columns list the identifiers of the pattern nodes containing parameters, and whose rows (r) contain the parameter values (v) for a specific path through the pattern. We refer to this data structure as a P-table, and it can be given in either a *verbose* (i.e., expanded), or a *compact* format (see Fig. 14b and Fig. 14c respectively). The procedure to create an instance argument is, effectively, to instantiate each *row instance fragment* (F), i.e., the fragment of the pattern beginning either at the root node of the pattern, or at a unique location—the *join point*, j—in the instance (I) at which an instantiated branch of the pattern is to be appended.

Hazard	Causes	Modes	Mitigation	Safety Requirement
HR1.4: Avionics system failure				
HR1.4.5: Flight critical system failure			MI1.4.5: Redundancy	RG1.4.3: Flight critical systems shall be dually redundant
HR1.4.5.3: Loss of Bank A	C1.4.5.3-1: Electrical overload		MI1.4.5.3-1: Wire sizing	RG2.1.7: Wires shall be sized appropriately to bear the rated load plus a safety margin
	C1.4.5.3-2: Mechanical decoupling		MI1.4.5.3-2: Fasterner design	RF.166: All fasteners shall have locking mechanisms
			MI1.4.5.3-3: Pre-flight checks	PF1.5: Pre-flight checklist shall include fastner checks to verify proper installation
HR1.4.5.5: Switch failure		MO1.4.5.5: Stuck open		

(a) Excerpt of a hazard table from the safety analysis of the Swift UAS.

	Parameter Type	*Hazard*	*Hazard Cause*	*Hazard Mode*	*Mitigation*	*Requirement*
	Data Node ID	G1	G2	G3	S3	G4
	Parameter ID	h1	c1	m1	m2	r1
		HR1.4				
Join Points	G1, HR1.4	HR1.4.5				
	G1, HR1.4.5	HR1.4.5.3			MI1.4.5	RG1.4.3
	G1, HR1.4.5.3		C1.4.5.3-1			
	G1, HR1.4.5.3			C1.4.5.3-2		
	G2, C1.4.5.3-1				MI1.4.5.3-1	RG2.1.7
	G3, C1.4.5.3-2				MI1.4.5.3-2	RG.166
	G3, C1.4.5.3-2				MI1.4.5.3-3	PF1.5
	G1, HR1.4.5	HR1.4.5.5				
	G1, HR1.4.5.5			MO1.4.5.5		

(b) Verbose *P*-table containing the (identifiers of the) data from the hazard table of Fig. 14a.

	Parameter Type	*Safety Requirement*	*Source*	*Implementation Allocation*	*Allocated Requirement*	*Verification Method*	*Verification Allocation*
	Data Node ID	G1	C1	C2	G2	S2	E1
	Parameter ID	r1	s1	a1	a2	v1	e1
Join Points		R1	S	A	AR1	VM11, VM12	[VA11, VA12], [VA22]
	G1, R1	R1.1, R1.2					
	G1, R1.1					VM1.11, VM1.12	[VA1.11, VA1.12]
	G1, R1.2	R1.2.1, R1.2.2			AR1.2		
	G1, R1.2.1					VM1.2.1	VA1.2.1
	G2, AR1.2				AR1.2.1	VM1.2	VA1.2

(c) Excerpt of compact *P*-table containing the (identifiers of the) data from a requirements table.

Fig. 14. Data required to instantiate the extended hazard directed breakdown pattern of Fig. 15a, and the requirements breakdown pattern of Fig. 15b.

Fig. 13 gives an abstract algorithm that illustrates the instantiation procedure; for the more detailed algorithms that address both the compact and the expanded *P*-tables, see [20] and [23].

6.2.1 Integration of Hazards and Requirements Analysis

We can create fragments of safety arguments from a hazard analysis, and the subsequent safety requirements analysis, e.g., as we have done from the functional hazard analysis (FHA) of the Swift UAS [15, 22].

In particular, an outcome of FHA is a hazard table containing a hierarchical list of hazards, their causes and/or modes, identified mitigation mechanisms, and the cor-

responding safety requirements. Fig. 14a shows an excerpt of such a table, which we transform into a *verbose P*-table (Fig. 14b), which is required to instantiate the *extended hazard-directed breakdown* (EHDB) pattern (Fig. 15a). In general, to map hazards/requirements tables to *P*-tables, each row of the former is mapped to a row of the latter so that parameter values are consistent with their types. Row increments occur based upon the join points in the pattern corresponding to the *P*-table or, for the verbose tables, when multi-row entries are to be mapped.

Here, note that the *P*-table only contains the identifiers of the entries of the hazard table of Fig. 14a to save space; in implementation, the *P*-table would instead contain the actual, complete data in addition to the identifiers. A result of the safety requirements analysis, which follows the FHA, is a requirements table (not shown here) containing a hierarchical list of the safety requirements identified in the hazard analysis, related verification methods, and the expected verification results. Again, we refer to (the identifiers of) the contents of the requirements table in a *P*-table (an excerpt of which is shown in a compact format in Fig. 14c), which we use to instantiate the *requirements breakdown* (RB) pattern (Fig. 15b).

The EHDB pattern (Fig. 15a) captures the implicit reasoning in the hazard table, i.e., how a claim of mitigation of the identified hazards can be hierarchically developed into claims of mitigating lower-level hazards, managing hazard causes and hazardous modes, and eventually linked to claims concerning the satisfaction of safety requirements arising from the identified mitigation mechanisms. Likewise, the RB pattern (Fig. 15b) explicitly represents the reasoning in the requirements table, i.e., how the claims entailed by safety requirements can be, further, hierarchically developed and linked to the supporting evidence produced from the specified verification methods. We do not give the descriptive specifications of the patterns here, and refer the reader to [23] for those details.[26]

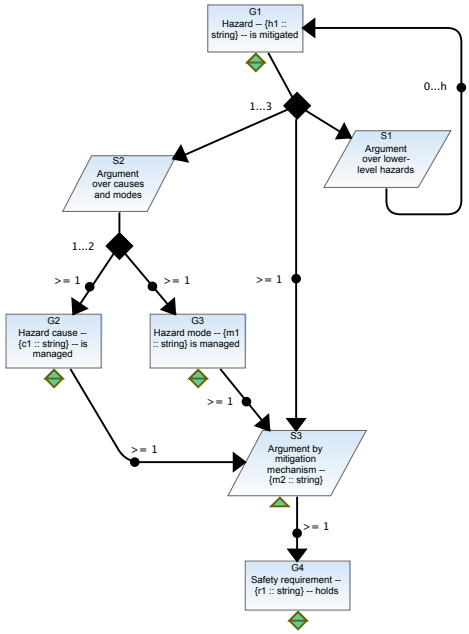
Additionally, note that the patterns and *P*-tables given here are specific to the particular hazard/requirements tables created during the safety analysis of the Swift UAS. During safety analysis, other forms of the tables may be defined that may contain additional items of data. In general, a domain-specific definition of an appropriate pattern—capturing the specific reasoning associated with a given analysis—and the corresponding mapping to the *P*-table is required.

Fig. 16 shows the arguments that we generate by instantiating the patterns in Fig. 15 using the *P*-tables of Fig. 14b, and Fig. 14c, respectively.

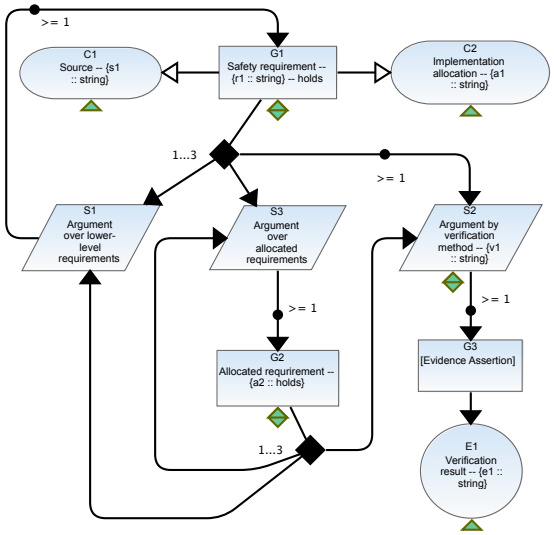
Upon comparing the leaf node of the EHDB pattern, and the root node of the RB pattern, we observe that the two nodes are identical. Thus, it is easy to see how the two patterns can be *composed* into a two-tier argument architecture so that, in the instance argument, the first tier addresses the identified hazards for a system, whereas the second tier assures that the corresponding safety requirements have been met.

As mentioned earlier, pattern instantiation has been applied by our colleagues at NASA Ames, to create an argument structure with well over two thousand nodes, to integrate over five hundred requirements for the safety assurance of a transport category, twin-engine aircraft model [12]. Additionally, we have applied this approach to integrate formal methods into safety arguments, which we describe next.

[26] The patterns shown here are more concise versions of those given in [23].

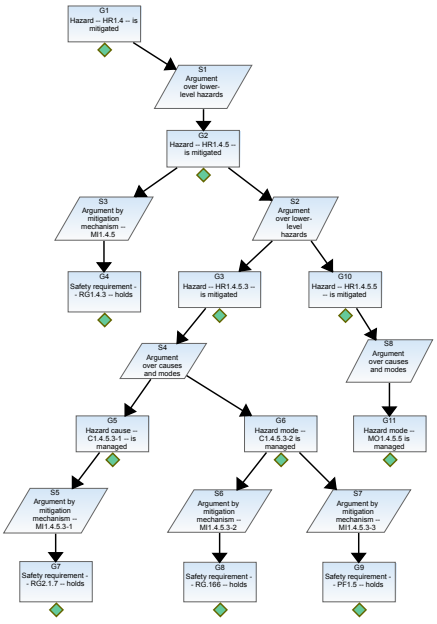


(a) Extended hazard-directed breakdown (EHDB) pattern.

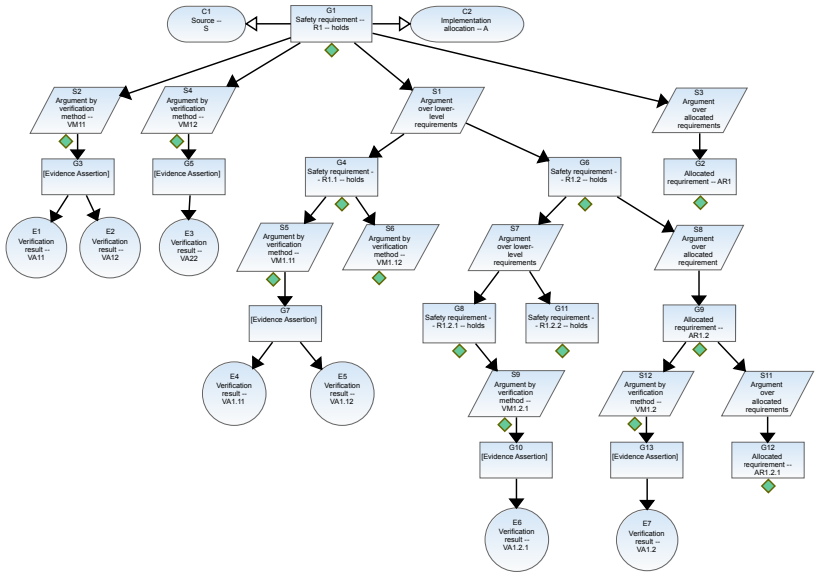


(b) Requirements breakdown (RB) pattern.

Fig. 15. Argument patterns, specified in AdvoCATE, for integrating hazard and requirements analysis in the safety argument for the Swift UAS.



(a) Instance of EHDB pattern (Fig. 15a) automatically generated using the *P*-table of Fig. 14b.



(b) Instance of RB pattern (Fig. 15b) automatically generated using the *P*-table of Fig. 14c.

Fig. 16. Automatically generated instance arguments.

6.2.2 Formal Methods Integration

Formal methods, such as formal verification, can be used conveniently in an argument to supply evidence. Traditionally, the approach is to use evidence nodes to refer to the artifacts created by a formal tool, and additionally supply justification for the relevance, and validity/efficacy of the evidence, e.g., addressing issues such as tool qualification, confidence, etc., for the overall argument. However, we are interested in using formal methods to create (fragments of) arguments, as opposed to simply evidence nodes. The idea is to use the reasoning underlying the formal method/tool itself to create an argument for the suitability of the results produced and to provide additional insight into the analysis beyond the existence of the evidence[27]. We also want to be able to invoke the formal methods tool from within AdvoCATE while constructing the argument. This entails several additional requirements:

- Formalizing informal nodes: that is, transitioning from informal to formal nodes within the argument;
- Mapping argument nodes into specifications, i.e., translating claims and assumptions into formal tasks in the language of the tool;
- Mapping formal verifications into argument fragments; and
- Integrating argument fragments into an overall argument.

The main idea is that the integration of a tool can be specified using patterns (Fig. 17). Thus, to integrate a tool we provide the patterns embodying the formal reasoning (used by the tool) and a mapping from the tool output (i.e., the verification artifacts) to the pattern parameters, via the appropriate *P*-tables. The dual mapping from arguments to specifications needs to map open goals (i.e., goals without support) to verification tasks, e.g., formal requirements, and assumptions in the scope of those goals to logical hypotheses.

We have used this approach to automatically generate and integrate an argument, which was used to support a software safety claim in the Swift UAS safety case [22], and which was created using the output of the AUTOCERT annotation inference tool [5], [29] applied to the verification of autopilot source code.

Fig. 18a shows a (screenshot containing a) fragment of the Swift UAS autopilot software safety argument. As shown, we develop a claim of correct implementation of the PID controller updates for the aircraft control surfaces (goal node G120), through an argument over each control surface (strategy node S120), into sub-claims concerning the correct implementation of controller for the aileron control surface (goal node G121) and the elevator control surface (goal node G122). To further develop the former—which is an informally stated claim—we first formalize the claim by invoking the *claim formalization* (CF) pattern [23] (through the *Formalize* option of a context menu, as shown in Fig. 18a). The user is then presented with a dialog (Fig. 18b) to interactively supply the values of the CF pattern parameters. The values of the parameters depend upon the language and logic being used for formalization.

[27] Note that by hierarchically abstracting such an argument into a closed hierarchical evidence node (see Sections 4.3 and 6.3.2), the result is an argument which is both structurally and semantically identical to that produced from the traditional approach of referring to the results of formal methods using evidence nodes.

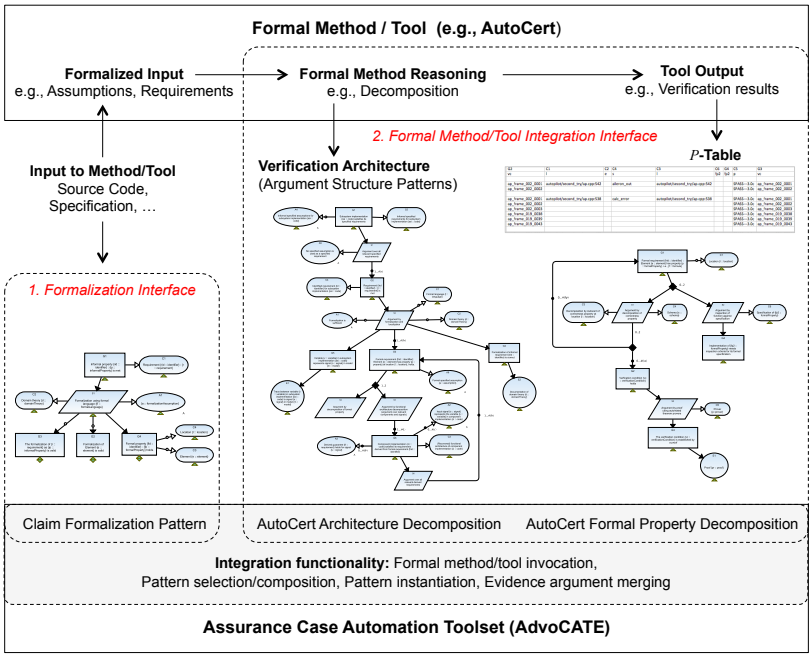
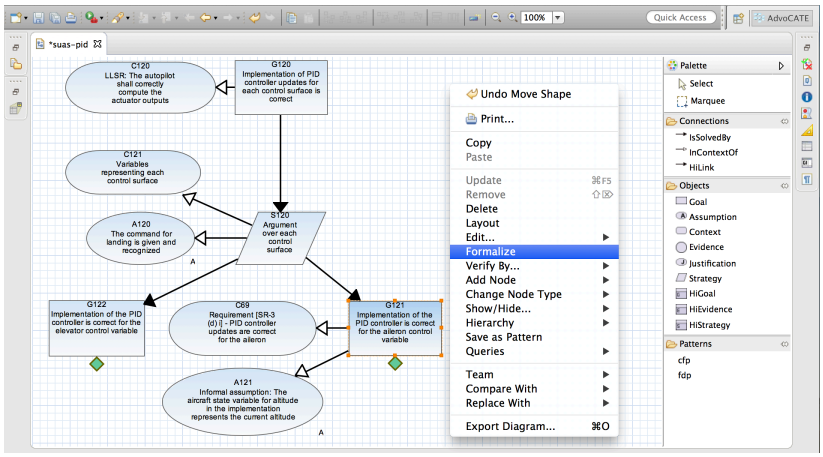


Fig. 17. Concept and architecture for integration of formal methods with safety cases.

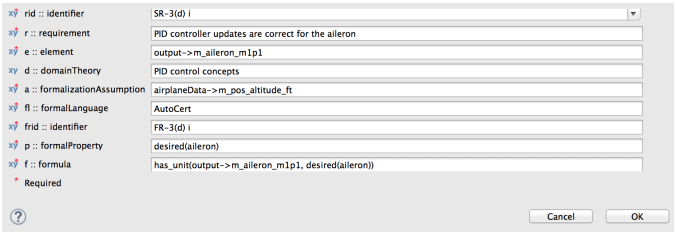
AdvoCATE then instantiates the CF pattern with the parameter values supplied, replacing the leaf node (G121) from which the CF pattern was invoked, with the instance argument (goal node G1 and all its children, in Fig. 19). As shown, the root node of the instance argument is identical to the informal claim from which the CF pattern was invoked, whereas the leaf nodes comprise the formalization of the informal claim (goal node G2), supporting claims about the validity of the claim formalization (goal node G4), and the element to which the informal requirement applied (goal node G3). Both the formal and informal nodes are updated with metadata relating the two.

Once we have formalized a claim (or if formal claims are already part of the argument), we invoke the verification tool from within AdvoCATE (see the *Verify By* option in the context menu, shown in Fig. 18a). Based on the metadata of the formal node, indicating what the corresponding requirement is, AdvoCATE calls AUTOCERT on a specific requirement in a designated specification file. The output produced from AUTOCERT is then

i) automatically converted into a data table consistent with the AUTOCERT *formal property decomposition* (FPD) pattern: a recursive pattern that defines the general structure of formal property decomposition in AUTOCERT [21]; after which



(a) Invoking the claim formalization (CF) pattern through the *Formalize* option of a context menu, as applied to an informal goal node (G121) in a fragment of the Swift UAS autopilot software safety case.



(b) Interface to interactively supply the parameters of the CF pattern.

Fig. 18. Screenshots of the AdvoCATE interface to formalize an informal claim.

ii) the FPD pattern is automatically instantiated (see Section 4.2). The instance argument structure produced encodes the reasoning and the evidence from AUTOCERT formal verification, which is then grafted onto the formal node from which the verification was invoked.

Fig. 20 illustrates this instance argument—i.e., the subtree with formal goal node G2 as root—but shows only one step in the verification, i.e., the decomposition of the formal claim in G2, into the VCs in goal nodes G124 and G126, and the lower-level formal property in goal node G6. The rest of the verification steps and and the corresponding nodes in the argument have been hidden[28] for better readability. The full argument can be seen in [21, 22].

In [22] we have described in detail how formal requirements verified using AUTOCERT can be transformed into argument fragments. Here, we have mainly de-

[28] AdvoCATE provides a *Show/Hide* feature—as shown by the eponymous option in the context menu in Fig. 18a—with which a user can selectively show and/or hide a node, paths to/from a node, and children of a node.

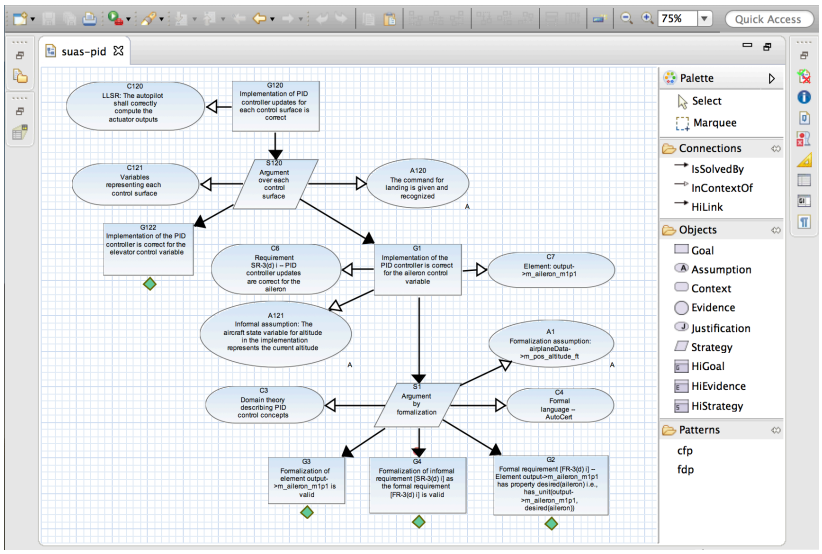


Fig. 19. Screenshot of AdvoCATE showing an instance of the CF pattern produced from interactive instantiation and appended to the invoking fragment.

scribed how AdvoCATE supports the process, in particular the steps followed in the AdvoCATE interface to invoke a formal verification tool and integrate the result into an argument. Although we have used AUTOCERT, as the formal method (and tool) being integrated, the approach is generic and extensible to other formal verification techniques[29] in a straightforward way.

6.3 Structural Abstraction

Earlier (Section 6.2), we described how we use AdvoCATE to automate the generation and assembly of large argument structures. In this section, we describe how we use AdvoCATE for structural abstraction of such arguments, in particular modular organization (Section 6.3.1), and hierarchization (Section 6.3.2).

We illustrate this functionality by applying it to the safety arguments of a real aviation system: a ground-based detect and avoid (GBDAA) capability used for the assurance of UAS flight transit operations [6]. In brief, the purpose of this safety argument is to provide assurance that *a*) the GBDAA capability meets its respective functional requirements, i.e., to detect intruder air-traffic in the airspace where UAS transit operations are occurring, and to facilitate the safe completion of any avoidance maneuvers required to mitigate a mid-air collision hazard; and, additionally, *b*) that

[29] As well as, more generally, to other formal methods paradigms, so that techniques such as formal refinement or program synthesis could be integrated, although that would require a different workflow.

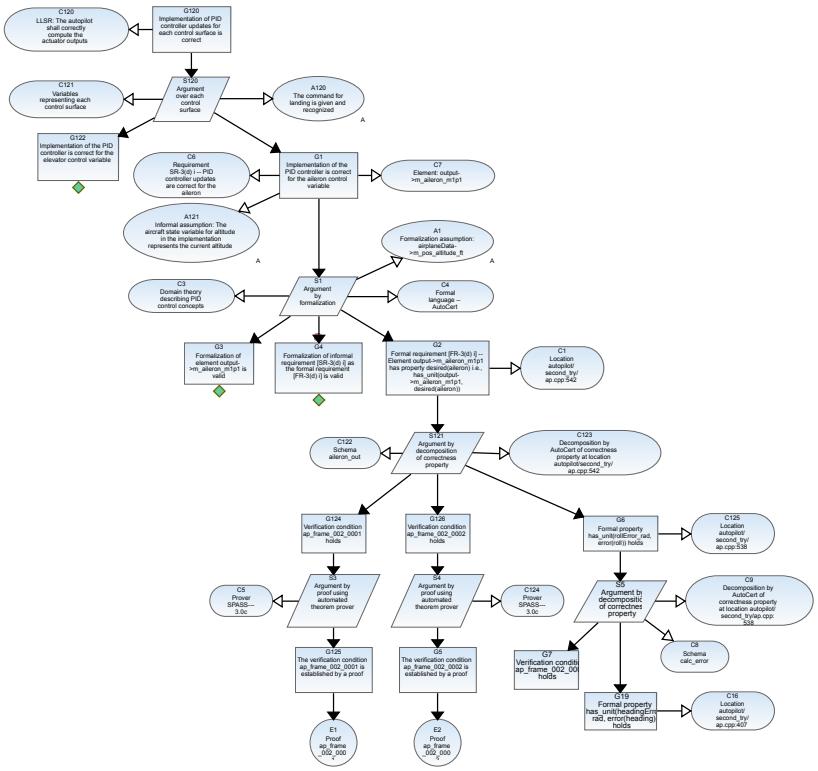


Fig. 20. One step in the AUTOCERT verification: one level of the instantiated verification argument, connected to the formal claim node with the rest of the levels hidden.

the introduction of this capability into the airspace system does not pose additional hazards to other airspace users.

6.3.1 Modular Organization

The currently implemented functionality allows users to create modular arguments, contractual arguments, module views, and a module hierarchy. To display and manipulate the latter, a dedicated *module explorer* panel is available (Fig. 21, top left) whose default content contains a *local root module* that corresponds to a project. The module explorer automatically recognizes and lists existing (or newly created) modular/contractual arguments. By default, all contents appear as siblings under the local root module. Users can then create, edit, and move modules (and the corresponding arguments) into a hierarchy as required.

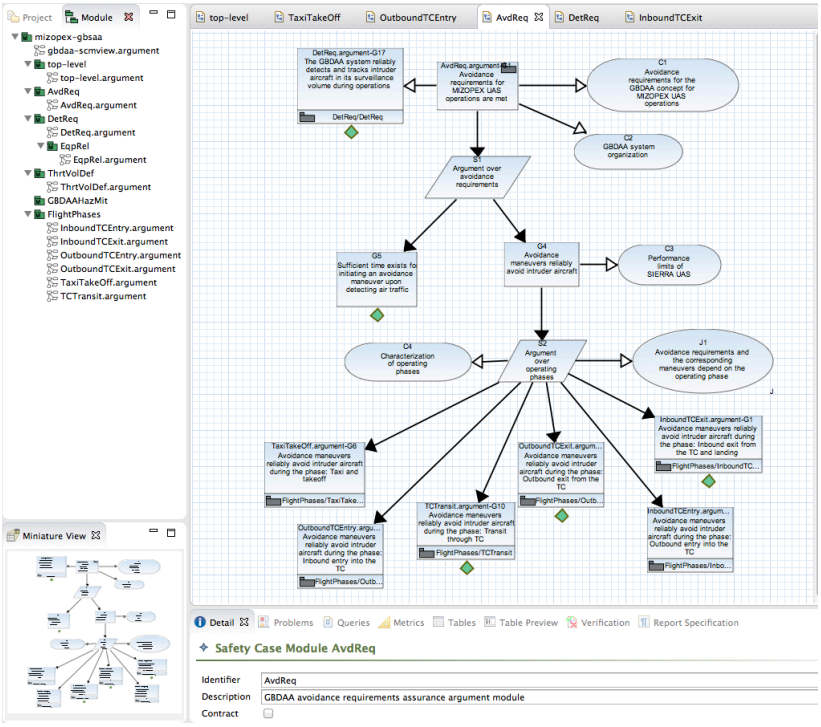


Fig. 21. Screenshot of AdvoCATE showing modular organization of arguments, with a module explorer panel highlighting the module hierarchy and a modular argument shown in the canvas.

Fig. 21 illustrates how we use AdvoCATE to organize the GBDAA safety argument in a modular way. As shown in the module hierarchy, the overall argument contains a *top-level* module, along with sibling modules that address:

- the GBDAA detection and avoidance functional requirements (module *DetReq* and module *AvdReq* respectively). The former further contains an argument of equipment reliability (module *EqpRel*);
- the mitigation of GBDAA hazards (module *GBDAAHazMit*);
- the adequacy of the *threat volume* of the airspace for mission operations (module *ThrtVolDef*);
- safety during the various flight phases (module *FlightPhases*).

Moreover, each module contains modular arguments that address a specific top-level claim. For instance, each of the six arguments in module *FlightPhases* i.e., *TaxiTakeOff*, *OutboundTCEntry*, *TCTransit*, *OutboundTCExit*, *InboundTCEntry*, and *InboundTCExit*, addresses a specific claim concerning that flight phase.

The module view (not shown) presents an overview of the safety argument architecture, representing the dependencies between the various modules and contract modules. For example, the *top-level* module invokes support from the modules *Av-*

dReq, *DetReq* and *GBDAAHazMit*. Likewise, module *AvdReq* is supported by module *FlightPhases*, and there are contextual dependencies between the modules *AvdReq*, *DetReq* and *ThrtVolDef*. Each module (or contract module) in the module view is linked to the corresponding diagram, and AdvoCATE provides navigability to the contained modular (or contractual) arguments. For example, in Fig. 21, the canvas tab labeled *AvdReq* shows a fragment of the modular argument contained in the GBDAA avoidance requirements assurance argument module (*AvdReq*), while Fig. 22 shows another fragment.

As shown in Fig. 21, the modular argument contains at least one away node of an appropriate type such that *i*) the module reference in the away node points to a module and modular argument defined in the module hierarchy/view; and, *ii*) the link type to that away node corresponds to the link type to the module given in the module view. Thus (as shown in Fig. 21 on the canvas labeled *AvdReq*), the module reference in the away node G17 refers to the module *DetReq*, and the node itself is referenced in context, which will result in an $\multimap$ link between the modules *AvdReq* and *DetReq* in the module view (not shown).

To maintain consistency between the module view and the corresponding modules, the user can either choose to have edits in the latter automatically propagated to the former (i.e., by regenerating the view) or to have the tool flag the view as being inconsistent with its modules and requiring a user edit. Conversely, there are various choices for how an edit to the view could be propagated to the modules but this is not currently implemented.

6.3.2 Hierarchization

Now we describe the application of hierarchical structuring, using AdvoCATE.

As mentioned earlier (Section 6.3.1), Fig. 22 shows a fragment of the GBDAA safety argument contained within the module *AvdReq* (Fig. 21). Since hierarchization is orthogonal to modularization, for clarity here we use a version of the argument that omits the modular elements. The main claim (goal node G7) in the fragment concerns the acceptability of the GBDAA function to safely avoid intruder air traffic. In brief, to provide assurance that this claim holds, the argument presents a chain of reasoning, together with evidence, regarding the procedural and technical implementations of the avoidance function. The former further includes reasoning over deconfliction and operator-specific procedures, whereas the latter considers airworthiness and equipage issues. Note that the structure of Fig. 22 has no hierarchical (or modular) abstraction and we consider it to be a *flat* safety argument.

Fig. 23 shows a possible hierarchization of the flat argument structure of Fig. 22, presenting an intuitive abstraction of the underlying reasoning, although other hierarchical organizations can also be created. As shown, we have hierarchically abstracted the goal node G8 (G9) and the entire subtree beneath it (as in Fig. 22) into the hierarchical goal HG2 (HG1), which has been shown in the *open* view, i.e., the contents are visible. In AdvoCATE, we achieve this result by simply selecting the node that will form the *local root* of the hinode to be created, e.g., G8 (G9), and invoking a *hierarchize* operation that automatically determines the contents of the hinode based upon

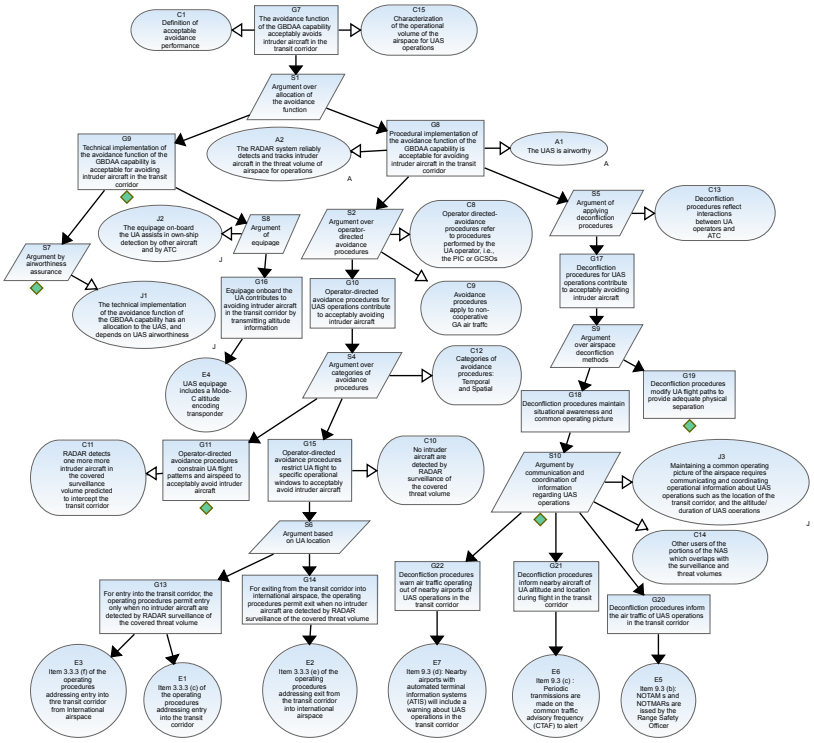


Fig. 22. *Flat* safety argument fragment, extracted from the safety case for a ground-based detect and avoid capability for UAS operations [6].

the well-formedness rules for hierarchization [27]. In this case, since no nodes are selected to delimit the *outputs* of the hinodes, the hinodes will extend to the leaves.

The fragment of the argument between, and including, strategy nodes S2 and S4 (in Fig. 22), is a chain of strategies that we can consider together and, therefore, abstract as a higher-level, hierarchical strategy; Fig. 23 presents this chain of strategies as the hinode HS1 (shown in its *closed* view) whose description reflects the combined strategy employed. Likewise, in Fig. 23, the hinode HS2 abstracts the fragment between, and including, the strategy nodes S5 and S9 (Fig. 22). In AdvoCATE, this result is achieved by first selecting the goal nodes that delimit the hierarchical node to be created (in the case of HS1, those nodes are the local root G8, and outputs G11, and G15) and, then, invoking the hierarchize operation. We can also create a hierarchical strategy in the same manner as a higoal, i.e., by first selecting the strategy node that will be the local root of the subtree to be enclosed, and then invoking the hierarchize operation. The hierarchical strategies HS3 and HS4, in Fig. 23, were created in this manner. The former encloses the subtree beneath and including the

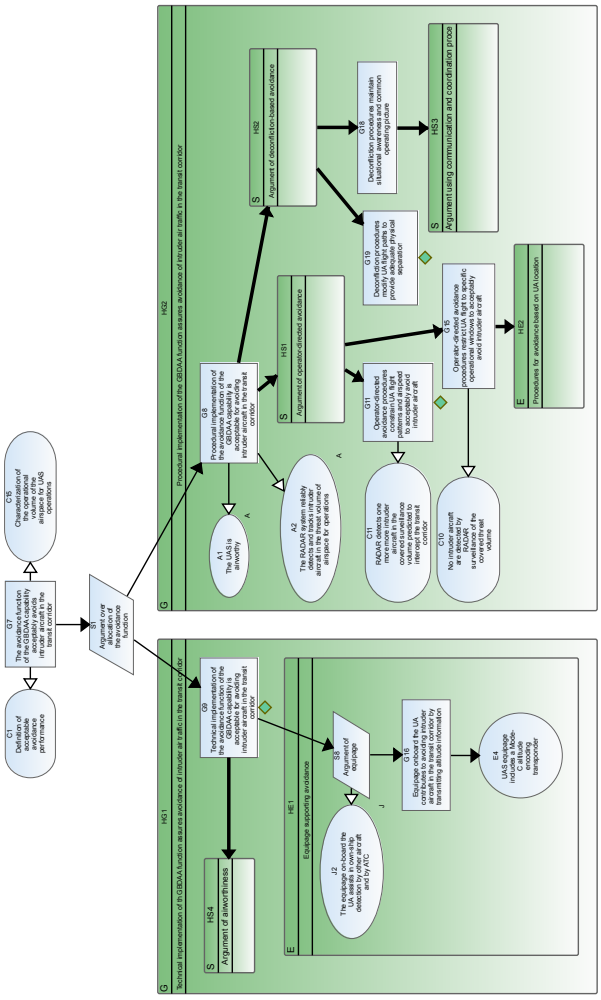


Fig. 23. A complete hierarchization of the flat argument structure of Fig. 22, showing containment as well as both *open* and *closed* hinodes.

strategy node S10 (see Fig. 22), whereas the latter abstracts the strategy node S7 and its justification. The hierarchical evidence node, HE2, is constructed in a similar way.

Each hinode description intuitively conveys either the overall conclusion to be drawn from the argument enclosed (in the case of a higoal or hierarchical evidence), or the combined strategy employed (in the case of a hierarchical strategy). Thus, for example, the higoal HG2 that abstracts the argument from goal G8 downwards would be used to conclude that a procedural implementation of the GBDAA function assures

that intruder air traffic would be avoided. Similarly, the hierarchical strategy HS1 concerns the usage of operator-directed avoidance procedures, and the hierarchical evidence HE2 reflects a collection of those procedures that depend on the location of the unmanned aircraft.

6.4 Queries and Views

Now we describe how we can use AdvoCATE to query a safety argument, so as to present views that reflect stakeholder perspectives.

For instance, certification processes in aviation usually require that traceability be demonstrated between regulations, identified safety hazards, and the various types of requirements (i.e., those addressing system safety, subsystems, components, and software). Thus, a query about traceability can serve, in part, to show compliance to the traceability requirements arising from the regulator's viewpoint. Similarly, from the perspective of software development, it can be useful to highlight software relevant claims in the argument, so as to evaluate their contribution to the identified safety hazards. Alternatively, from the perspective of safety case development, queries can also be used to support the argument development process, in particular argument assessment (Section 2.3, and Fig. 3), e.g., by identifying parts of the argument that, though complete, might not engender sufficient confidence. For example, goals associated with high risk may need to be supported by particular forms of evidence. We can then use an appropriate query to identify those argument fragments that do not meet these criteria.

To illustrate the query functionality and show its utility, we apply it to a fragment of the Swift UAS safety case (shown as a bird's eye-view in Fig. 24). In brief, the root node of the fragment addresses the mitigation of a specific safety hazard—i.e., *unanticipated nose pitch down during descent and landing*—that can result in a loss of the aircraft and damage to the runway. The argument develops the root claim of hazard mitigation into sub-claims concerning the various contributory system functions, including software/hardware, components, and operations, which are then linked to the evidence, e.g., available from experimental data, procedures, and verification activities. We augment the argument nodes with metadata, e.g., user-defined enumerations (given in a domain-specific grammar [18]), which we can reference in queries.

Fig. 25 shows a screenshot of the AdvoCATE interface, highlighting an editor in which to specify queries, as well as the verification environment (described later, in Section 6.5). The query editor (see the *Queries* panel in Fig. 25) provides a library of pre-defined queries. Users can specify a customized query in the AdvoCATE Query Language (AQL), store those queries as drafts, and add well-formed *named queries* to the query library. For improved readability and ready access to the AQL keywords, the editor provides syntax highlighting and auto-completion. As shown in Fig. 25, the query to be executed has been given as the AQL expression:

```
isGoal & hasAttribute(regulation) |
isBelow(isGoal & hasAttribute(regulation)).
```

This query attempts to *a*) identify those goal nodes in the argument with the attribute 'regulation'; and, additionally, *b*) show the fragments that have the located goals as

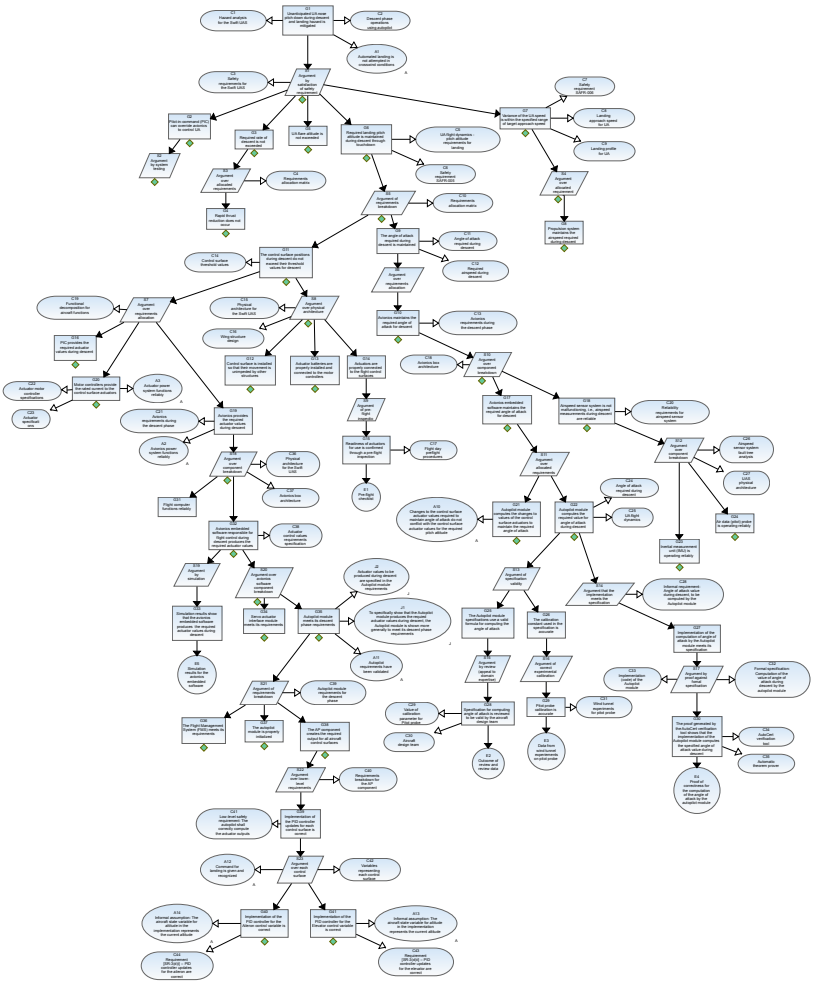


Fig. 24. Bird's eye view of a fragment of the Swift UAS safety case, addressing the mitigation of a *nose pitch down* hazard [18].

root. Such a query could be used, for example, to determine all the parts of the argument that address the concerns arising from regulations (or, in general, standards, guidance documents, etc., when argument nodes contain the corresponding metadata).

Others [68] have also recognized the need to relate traceability to safety arguments, defining a specific *traceability information model* and a mapping to GSN arguments. Whereas this approach utilizes specific types of metadata and a static

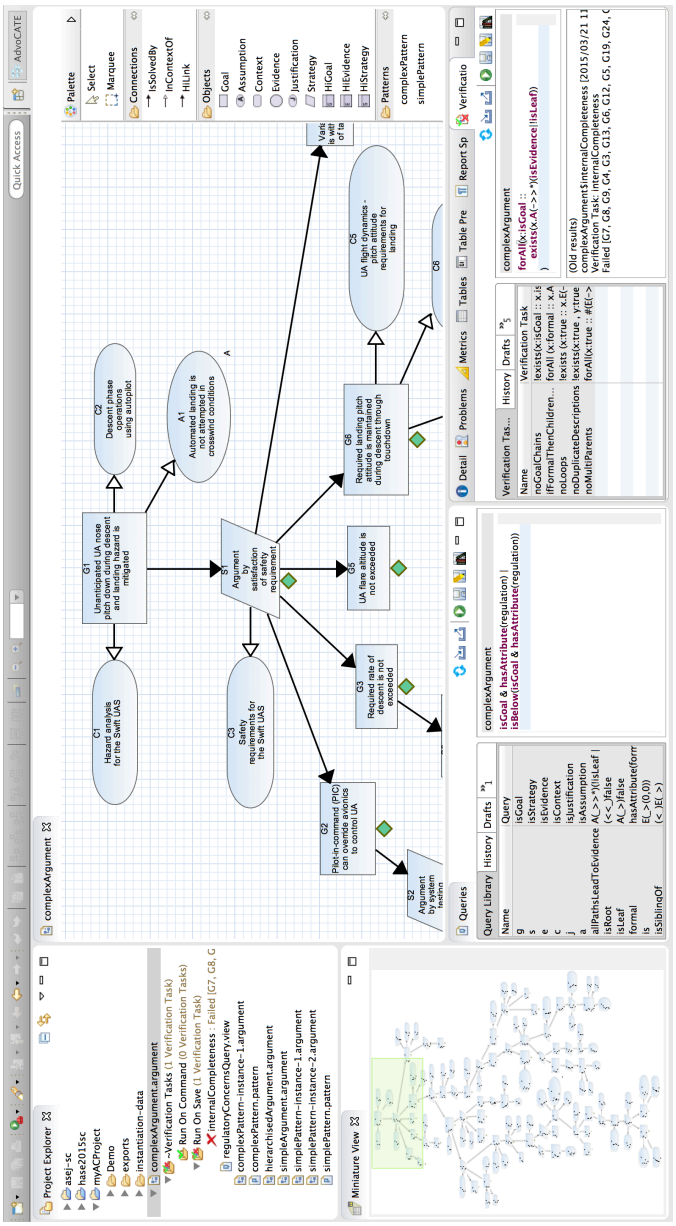


Fig. 25. Screenshot of the AdvoCATE interface for specifying queries (in the *Queries* panel on the bottom left), and for property verification (in the *Verification* panel on the bottom right). Note that in both panels the relevant editor gives the name of the argument (complexArgument) upon which the query/verification is executed; here, that also happens to be the argument shown on the canvas.

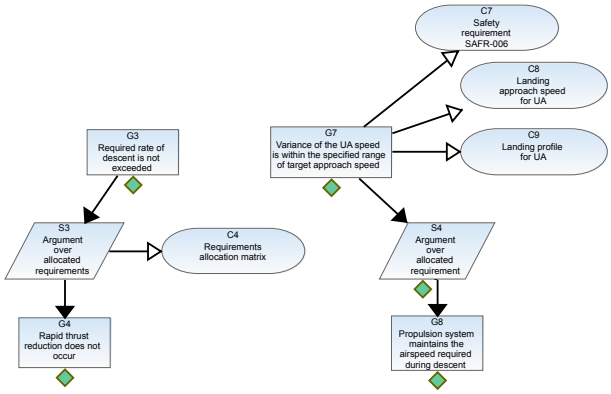


Fig. 26. View produced by executing the query shown in Fig. 25.

mapping to particular argument node types, metadata within our tool is user-defined, and can be associated with any argument node.

After executing the query, AdvoCATE produces a view (Fig. 26)—showing the two fragments of the argument (of Fig. 24) that satisfy the query—from which we can infer that *i*) there are claims in the structure that reference regulatory requirements, but that *ii*) they are yet to be fully developed and supported by evidence. To determine the exact extent of how the regulations are met, an assessor could navigate to, and examine, the external documentation referenced from the nodes shown in the view.

Note that this query is rather general, although intentionally so (since we want to see *all* the parts of the argument that concern regulations). We can give more specific queries by including metadata parameters. Thus, if we wanted only to determine whether the regulation concerning aircraft performance during landing—in particular approach speeds (14 CFR §23.73)—were addressed, we would alter the query to state:

```
isGoal & hasAttribute(regulation(CFR14-Part23-73)) |
isBelow(isGoal & hasAttribute(regulation(CFR14-Part23-73)))
```

which would produce only the right half of the view shown in Fig. 26, i.e., the fragment with node G7 as root. Thus, we can specify generic or detailed queries that best capture, and locate, the information sought.

Although the view in Fig. 26 appears to indicate that the fragments are disconnected, in fact, they are not. To highlight their connectivity, AdvoCATE introduces a new node type (not shown in Fig. 26, see [18] for an example)—i.e., a *concealment* node, or $\mathscr{C}$-node—only visible in views. A $\mathscr{C}$-node is a collapsed representation of all the nodes that do not satisfy the query, and is annotated with the number of hidden *core* nodes [18]. Displaying $\mathscr{C}$-nodes is a user-defined setting in AdvoCATE and, should a user choose to display them, they can be shown when the nodes that do not satisfy a query have either only incoming links, only outgoing links, or both.

In general, AdvoCATE stores views as a special property of the diagram to which the query is applied, in particular as two lists in the diagram file itself: *a*) all the view

names associated with the diagram; and, correspondingly, *b*) the query that maps to each name. In the interface, views appear by name as sub-items under the corresponding diagram in the project explorer (see Fig. 25, where the view produced from the query shown has been listed). We have implemented some additional usability features, such as the ability to open multiple views simultaneously in separate tabs, i.e., multiple canvases. Users can save changes either to the argument structures, the queries, or both. When any change is made either to the source diagram or a view, it is reflected in all views and in the original diagram.

In summary, we can use AQL to specify combinations of structure and metadata to produce views that address domain specific scenarios. We can also use AQL to specify more complex queries that operate on the structure alone, to produce meaningful views [18]. For example, querying to locate all goal nodes from which all paths lead to an evidence node (i.e., *completely developed* fragments), will produce a view that is identical to the argument structure to which the query is applied, if the argument is *internally complete*. However, as we shall see subsequently (Section 6.5), it is useful to be able to *specify* and *verify* such argument properties as well.

Extending Queries for Report Generation: Report generation is a common feature in model-based development, and many frameworks offer mechanisms for specifying reports using combinations of static and dynamic text, where the dynamic text is instantiated from a variety of artifacts, such as models, code, and test results. Matlab [55], for example, uses model queries in combination with structure and style elements to generate reports.

We have developed a prototype safety case report specification language, SCRbL, based on an extension to our query language, AQL. Fig. 27 shows the methodology for report generation. The key components are *templates*, specifying the organization and format for a report, and *queries* specifying the content. Templates combine boilerplate text with extended queries to extract information from the safety case. We omit the details here.

6.5 Verification

The overall goal for argument verification is to support our argument development process (Section 2.3), in particular the argument analysis and improvement activities (Fig. 3). Our current implementation can verify argument structure properties representing both *a*) specific structural constraints, i.e., syntactic checks (see Section 2.3.5 for some of the types of argument properties being addressed); and *b*) semantic checks, based on the metadata attached to the nodes. Note this is a form of *lightweight* semantic checking, in contrast with approaches which encode the argument itself in a formal machine checkable language [69]. In brief, the verification language extends AQL with a more expressive logic, including existential and universal quantification. As mentioned earlier, Fig. 25 shows a screenshot of the AdvoCATE interface including its (query, and) argument verification environment. Fig. 28 shows the latter in more detail. As shown, the verification environment provides a property specification editor in which we use APL to specify the properties to be

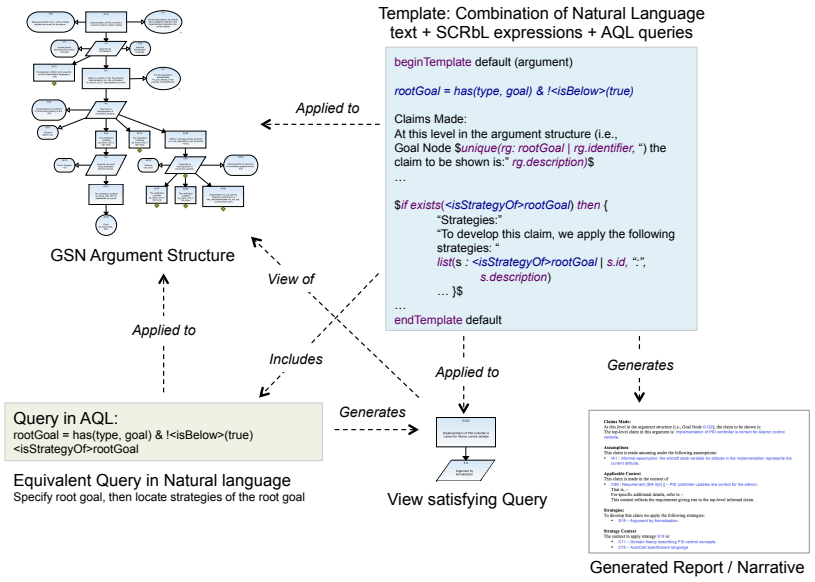


Fig. 27. Methodology for generating safety case reports using AQL queries and templates specified using SCRbL.

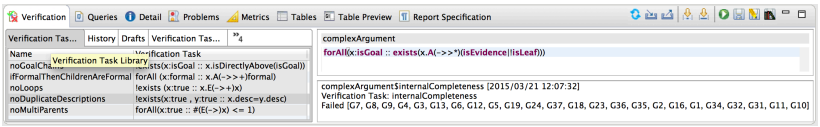


Fig. 28. Screenshot of AdvoCATE verification environment.

verified on a chosen argument structure. As in the query editor, the property specification editor provides standard editing functionality such as keyword suggestion and auto-completion during property specification, syntax highlighting, and an error indication if the property is not well-formed. Properties that are not well-formed preclude invocation of the verification engine.

When a well-formed property is verified, upon successful verification, a *Passed* status is displayed, whereas a failed verification causes a *Failed* status to be displayed along with the counterexample nodes that caused the verification to fail (Fig. 28). The verification environment additionally provides library properties (shown in the *verification task library* sub-panel of Fig. 28), functionality to specify and save properties incrementally as drafts, a history of the properties executed, as well as links to the library of metrics and queries (not shown), which can also be used in verification. Furthermore, verifications can be saved relative to an argument structure (see the *project explorer* panel, in Fig. 25), and can be accessed directly from the project. The

idea is that property verification can be invoked either every time that a change is saved (i.e., *run on save*), or on user demand (i.e., *run on command*).

To illustrate the application of the verification functionality, we verify whether the Swift UAS argument fragment of Fig. 24 is internally complete, i.e., that all goals in the argument eventually have a path to an evidence node. We specify this as the property forAll(x:isGoal :: exists(x.A($->>$*)(isEvidence | !isLeaf))), which states that for all goals x there exists a node that is $\rightarrow$-below x, for which every node reachable by $\rightarrow^*$ is either evidence or not a leaf.

Fig. 25, and also Fig. 28, show that the argument structure (of Fig. 24) fails verification of this property, and also lists the goal nodes in the structure for which the property does not hold, i.e., the counterexamples to the universal quantification. Presently, there is no navigability from counterexamples to the argument structure, but we intend to introduce this functionality in future.

6.6 Computing Safety Case Metrics

Safety case metrics are a mathematical specification of the particular questions that must be addressed to respond to specific measurement goals, and which are determined by computation on the argument structure. Together with an *interpretation* model, metrics provide a convenient mechanism for decision making by summarizing the state, and the key properties, of a safety case during its evolution. Since measurement goals, the corresponding questions, and the interpretation models are usually defined and tailored to a particular application, domain, or project, there is a need for user-defined specification and computation of safety case metrics.

Although we will not discuss metrics and their treatment in detail here, we have defined a number of *base* and *derived* metrics that express, respectively, a value assignment to directly measurable safety case properties, e.g., size, and indirectly measurable properties, e.g., coverage of certain kinds of claims. In our earlier implementation of AdvoCATE [26], those metrics were not user-modifiable and were hard-coded. In our current implementation, we have extended AQL to support a user-defined computation of safety case metrics[30]. Additional syntax is introduced in AQL to support counting, integer literals, and basic arithmetic operations. For example, we can define a metric: developedClaimRatio $=$ #(isGoal & isTBD)/#isGoal specifying the proportion of undeveloped claims in the argument. In other words, it provides one numeric measure of the extent to which an argument has been developed, and can serve to support decision-making at the milestones at which the various processes during safety assurance synchronize (see Section 2.2).

We note that one of the criticisms against the safety case approach is the lack of a measurement basis [80]. Although the metrics being computed in AdvoCATE using our approach do not directly address this problem, we believe it provides preliminary steps towards bridging that gap.

[30] In fact, we can also leverage the use of metrics computation during property verification (Section 6.5).

7 Discussion

In this section, first we describe related work in tool support for assurance cases (Section 7.1). Then (in Section 7.2), we reflect upon the satisfaction of the needs and requirements motivating this work (given earlier in Sections 3.1 and 3.2). Finally, we present our future plans for enhancing AdvoCATE (Section 7.3).

7.1 Related Work

As mentioned earlier, practitioners can choose from among a variety of tools to create structured safety arguments using GSN and/or CAE diagrams.

The *Safety Argument Manager* (SAM) [57, 82] and its various versions represent, perhaps, the very first set of tools for creating and managing safety arguments. The initial incarnation of SAM based its structuring principles on the Toulmin model of arguments [70]. Subsequent versions of the tool focused on developing a goals-based representation for structured arguments (representing the precursor to GSN), and managing the interrelations with fault tree analysis (FTA) and failure modes and effects analysis (FMEA). A number of tools for creating and managing safety cases have since emerged (including ours), several of which have been developed through research projects, which we will categorize as *research* tools. Others tools have been developed for commercial purposes, which we categorize, accordingly, as *commercial* tools. We describe each category next.

7.1.1 Research Tools

Several research tools have used the Eclipse framework for implementing functionality to create and manage safety case argument structures. For example, *ACEdit* [30] is an open-source, Eclipse plugin[31] that provides a straightforward implementation of the GSN standard and the Argumentation Metamodel (ARM). Likewise, the *SafeEd* tool[32] is another Eclipse plugin for creating GSN safety arguments, supporting only the core (non-modular) GSN. However, SafeEd is distinguished by its functionality of translating arguments into description logic, and automated consistency checking through an integrated reasoner. To our knowledge, neither tool provides any additional automation features, beyond those that are built into the Eclipse framework and extended, as a consequence, into its plugins.

The *AutoFocus3* (AF3) framework [79] is a set of model-based tools[33] one of which supports creating GSN arguments. Thus, the AF3 framework, itself, is a toolchain providing integrated support for requirements specification, modeling and simulation, code generation, formal verification, and test-case generation for distributed, reactive, embedded software systems. Models created within the framework can be linked to elements of the argument also created within the framework.

[31] Available at: https://code.google.com/p/acedit/

[32] Available at: http://cs-gw.utcluj.ro/ adrian/tools/safed/gsn/gsn.html

[33] Available at: http://af3.fortiss.org/

The *CertWare* workbench [4] is an open-source tool[34] for creating safety cases, supporting a variety of notations and metamodels. Thus, CertWare provides a facility to create diagrams in both GSN, CAE, and the EUROCONTROL GSN [33], which is, itself, a variant of the core GSN. Additionally, the argument data stored is compliant with the ARM, and the software assurance evidence metamodel (SAEM) (which have been now combined into the *structured assurance case metamodel* (SACM) [60]). CertWare also facilitates computing effort and project management metrics, rather than metrics on the safety argument, as in AdvoCATE (see Section 6.6).

The *D-Case* editor [56] is an open-source Eclipse plugin[35], which provides functionality to manually create argument structures and patterns. Arguments can incorporate *monitoring nodes*, a sub-class of evidence nodes which are linked to monitors of target systems and whose status is updated automatically to indicate whether target values remain within a specified range. The editor has also been connected to the *Agda* functional programming language and proof assistant, to provide features for automated consistency checking [69]. The methodology underlying the latter is to create a *formal assurance case* as a formal proof which can then be imported into the D-Case editor, so that consistency checking of the argument amounts to type checking the proof in an appropriate theory.

The *Evidence Confidence Assessor* (EviCA) [59] is an Eclipse plugin that supports the manual creation of GSN arguments, and purports to auto-generate the corresponding confidence arguments. The core approach is to manually annotate argument structures with so-called *assurance claim points* [46], which correspond to predefined confidence argument patterns. The core elements of these patterns are what the developers refer to as *confidence factors*. Pattern instantiation amounts to an interactive specification of the values of the relevant pattern variables, to create instance arguments. The tool additionally implements the evidential reasoning algorithm [83] to aggregate user-supplied beliefs and weights towards a specification of confidence in the arguments created.

Tools for creating and managing safety arguments, which do not (or only partly) use the Eclipse framework in their implementation are also available. For example, the *ACCESS* tool [66] is a Microsoft Windows application that integrates Microsoft Visio, as the application to create and edit GSN argument structures, and Microsoft Word, to view the related documents. In addition, it provides a number of customized interfaces to create argument summaries (i.e., descriptions associated with a selected subset of the argument nodes), record author or reviewer comments as annotations, and for argument inspection.

The *DiaSAR* [84] tool has been created with the primary purpose of supporting a *dialectical* model for reviewing GSN safety arguments. The tool provides an editor for specifying safety arguments using core GSN, along with some notational extensions to represent counter-arguments. In addition, the tool provides a means to track the progress of a review, i.e., whether an argument element has been accepted, withdrawn, challenged, or questioned, as well the dialog between the review participants (and its history).

[34] Available at: http://nasa.github.io/CertWare/

[35] Available at: http://www.dcase.jp/index_en.html

The *Evidential Tool Bus* (ETB) [14] provides a tool integration and workflow scripting framework to manage the claims and evidence generated by formal tools, in support of assurance activities. Specifically, the ETB provides a mechanical approach to managing the evidence both generated in support of tool-specific inferences, and created for the claims where formal tools have been utilized.

The *Workflow Engine for Analysis, Certification, and Test* (WEFACT) framework provides a toolchain[36] that connects, in part, the DOORS requirements management tool[37] and the D-Case Editor to generate GSN arguments for process compliance. The underlying methodology relies on model transformation, wherein *i*) a DOORS plugin first transforms a process model into its corresponding argument, i.e., essentially, a DOORS file that contains a listing of the argument elements together with the associated links, using a defined set of transformation rules [38]; and *ii*) the argument is then exported into a format compatible with the D-Case editor, which presents it as a GSN argument structure diagram.

The University of York, UK, provides a Microsoft Visio plugin[38] that provides a diagramming capability to create GSN argument structures.

The *Resolute* tool [37] provides both a language and a capability for creating assurance arguments based on architectural models. The language component of the tool provides users with the functionality to formulate claims and rules for justifying those claims, based upon which an argument structure can be generated. Resolute has been implemented based on the Eclipse framework.

Beyond the tools mentioned here, the literature contains allusions to other tool implementations: for example, of an approach for the *automated compilation of safety argumentation* (essentially a hierarchy of safety requirements) from work products of automotive functional safety development processes [2]; of a *weaving model* for automated pattern instantiation [45]; and, of a common information model for managing both arguments and evidence [62]. There is little information available on these tools, their actual capabilities, and their applicability, besides their provision of specific functionality in support of creating assurance cases.

7.1.2 Commercial Tools

The *Assurance and Safety Case Environment* (ASCE)[39] is a mature toolset for safety cases (and possibly also amongst the most widely used by many practitioners).

ASCE provides its safety case creation and management services through *schemas* and *plugins*. ASCE schemas provide the capability to recognize a variety of notations, including the (core and modular) GSN, CAE, fault tree notations, etc. To manipulate these notations, ASCE plugins provide a variety of functionality. For example, linking to (and embedding) a wide range of documents; listing argument contents in tabular form; customized report generation; support for creating and editing views of safety arguments, etc. We note that the underlying notion of view in ASCE differs from that

[36] See: https://www.ait.ac.at/en/research-fields/verification-validation/methods-and-tools/wefact/

[37] Available at: http://www-03.ibm.com/software/products/en/ratidoor

[38] Available at: http://www.goalstructuringnotation.info/archives/41

[39] Available at: http://www.adelard.com/asce/

supported in AdvoCATE [18]. Specifically, in ASCE, argument fragments must be manually selected, rather than determined through queries, as in AdvoCATE, after which those fragments can be viewed, edited and saved independently. ASCE keeps the views of an argument consistent with the original diagram. The *dynamic narrative region* (DNR) plugin provides users with the capability to extract and manage content from a variety of external sources, to embed the extracted content into the arguments being created, and subsequently update the content by validating consistency with the source documents. ASCE additionally provides functionality for change tracking via its *difference tool*, and support for independent configuration management of the argument and associated evidence.

Astah GSN[40] is a modeling tool that implements the SACM and the GSN. Essentially, it is diagramming tool with some automation features—e.g., a one-click export of a subtree in the argument into a module—primarily addressing usability needs, rather than a syntactic and/or semantic manipulation of the argument.

The *NOR-STA Assurance Case Tool*[41] is a web-based application whose focus is on creating the logical structure of an assurance argument, rather than its graphical construction using notations such as GSN. Primarily, it offers its users the ability to manually instantiate an argumentation model—the so-called NOR-STA argumentation model—comprising much of the same elements as core GSN along with some additional elements. Effectively, the arguments constructed using the NOR-STA assurance case tool are tree-like hierarchical structures. Nevertheless, the tool also encodes a mapping from the NOR-STA argumentation model elements to GSN, due to which instances of the argumentation model can be exported as GSN diagrams. The tool additionally offers capabilities to generate reports, and provide argument *assessments* using subjective logic [50].

The *Safety Case Toolkit* (SCT) [53], purports to provide a comprehensive system for developing, managing and presenting large-scale safety arguments as a set of services integrated into the Eclipse framework. SCT exports safety cases as websites, which are linked to the underlying safety artifacts, i.e., GSN arguments, text-based descriptions, figures, and linked files that are to be contained in the safety case. SCT provides a *build system* that provides automation support, primarily, to create the safety case website whose index is mechanically extracted from the specified argument structure and the linked files. In addition, SCT provides additional features in terms of auto-layout of the arguments, typed metadata, versioning support, and report generation. Finally, it also provides support for argument review, in part, via a customizable *focus* mechanism, that displays a selected node in the argument and a user-defined number of levels of descendants.

The *TurboAC* assurance case tool[42] implements a subset of the core GSN—i.e., goals, strategies, and solutions—and also provides a tabular representation of the created arguments. Additionally, it provides an interface to create fault trees, risk-traceability matrices, and to document FMEAs, with primary application to medical

[40] Available at: http://astah.net/editions/gsn

[41] Available at: https://www.argevide.com/en/products/assurance_case

[42] Available at: http://www.gessnet.com/products

device safety assurance [32]. The tool also provides an internal linking mechanism so as to reference the resulting analysis in the arguments created.

Safety.Lab [61], is a prototype tool whose focus is on so-called model-based safety analysis and it purports to provide support for rich, deeply integrated models of various safety relevant artifacts including requirements, hazards, fault trees and architecture. One of the capabilities of this tool is to create GSN argument structures that link to the safety artifact models that it creates and manages.

7.2 Reflection

To show traceability from the AdvoCATE implementation to the needs motivating this work (Section 3.1), we recall that the corresponding requirements (Section 3.2) state the provision of specific functionality. In particular, the implementation provides a formally-founded framework to:

- create and instantiate argument patterns from the artifacts produced during system development, in particular hazards and requirements tables, and formal methods (Section 6.2), towards addressing the requirement to support the automated creation and assembly of safety argument fragments (Fig. 4, requirement R1). Moreover, patterns and their automated instantiation provide a mechanism for evolving arguments and maintaining consistency when system development artifacts change.
- organize argument structures in a hierarchical and modular way (Section 6.3), to satisfy the requirement to provide complexity management features (Fig. 4, requirement R2).
- specify queries on argument structures, and produce views (Section 6.4) using the AdvoCATE Query Language (AQL). In general AQL forms the core of a suite of analysis languages to both *i*) enable improved comprehension of safety arguments by stakeholders (Fig. 4, requirement R3), through a view-based presentation of the argument, and *ii*) support evaluation processes (Fig. 4, requirement R4), through property verification and computing metrics.

Although we have only briefly touched on the formal foundations of the tool (Section 4), it is worth noting how this provides a basis for the automation features we have described. The semantics are based on a family of graph-based structures, with essentially no constraints on the contents of the nodes, themselves, allowing informal text. This can be contrasted[43] with (but is not incompatible with) work on formal assurance cases [63, 69] where assertions of the safety case are drawn from a formal logic. Thus, our emphasis on formalization is to provide robust, tool-based automation support for creating arguments.

So far as we are aware, the automation features of AdvoCATE as presented in this paper represent functionality not provided by other contemporary tools described earlier (Sections 7.1.1 and 7.1.2).

[43] This distinction has sometimes been misunderstood in the literature [41].

7.3 Future Work

Several avenues exist to extend and further develop the automation capabilities currently implemented in AdvoCATE, which we now describe.

7.3.1 Extending the Integration of Formal Methods

Although we have developed a generic mechanism for integrating formal verification tools, the integration of the individual tools, themselves, is currently hard-coded using the plugin mechanism. For full extensibility, we must identify the data that needs to be associated with a tool and how it should be user-specified. For example, tool integration could be specified as embeddings into the three argument pattern types: formalization, decomposition, and solution (i.e., divide and conquer), giving the architecture of the verification, along with mappings from argument nodes to tool input (i.e., specifications and syntax for calling a tool). More generally, this approach could be extended to provide a systematic way of integrating other software engineering artifacts, such as tests, and the corresponding traceability artifacts.

7.3.2 Pattern Composition

We implicitly use a notion of sequential composition of patterns. In fact, other ways of combining patterns can be defined, and we have also developed a notion of parallel composition (not given here) to create complex patterns (such as for requirements breakdown shown in Fig.15b) from simpler patterns. This leads to a notion of argument architecture, based on pattern composition. Pattern metadata could be drawn from an ontology, which would offer several advantages. For example, using an ontology tool together with an automated import mechanism would save some effort, while enforcing constraints from the ontology in the argument would provide a way of specializing a generic pattern for use in a particular domain, as well as validating that its application does not violate domain-specific constraints.

7.3.3 Combining Modularity, Hierarchy, and Patterns

There are several interesting lines of future development of hierarchy, modularity, and patterns in safety cases. Firstly, since our current theory of hierarchy only accounts for the core GSN, one key task is to extend the notion of hierarchy to also account for patterns and modules. Likewise, our theory of modules has not yet considered the various hierarchical nodes. Thus, a complementary task is to extend modules to account for hinodes. Secondly, we would like learn potential hierarchical and modular structure, e.g., using metadata, after which one might be able to automatically abstract hierarchical and/or modular patterns from existing safety cases.

7.3.4 Enhancing Queries and Views

Currently, the scope of a query is a single argument structure. We intend to develop suitable interfaces to linked artifacts, which will allow us to query the entire assembly of artifacts comprising a safety case. The current query/view mechanism is also

limited to the core GSN language, but AdvoCATE implements a number of extensions, in particular, for modules, hierarchy, and patterns. We plan to investigate the interaction of queries with these extensions. Additionally, as currently implemented, the source diagram and resulting view are each currently a single diagram. However, modular GSN allows multi-diagram safety cases, suggesting a need to define an appropriate notion of *multi-diagram view*. Another interesting direction to pursue is the interaction between queries and verification: so-called *ontology-mediated query languages* [7] combine querying with reasoning over an ontology, which may provide greater scope to formulate queries that address argument validity.

7.3.5 Extending Support for Argument Evaluation and Improvement

Among the major challenges in using safety cases during certification, is evaluating the argument presented. This can be considered as analogous (or complementary) to the activity of argument analysis (Section 2.3.5), although existing techniques such as phased inspection [42], and safety argument review [51] render it a largely human-centric verification activity. We believe that AdvoCATE can provide tool-based automation support here, e.g., through the integration of Bayesian reasoning for the assessment of confidence [25].

Additional research in terms of both methodology and automation support, is also required for improving an argument after it has been evaluated, in particular with respect to the argument improvement activity (Section 2.3.6). So far, the metrics we have implemented compute properties of individual arguments, but could be extended to modules. The concepts of coupling and coherence are well-known quality metrics for modularization, but the analogous notions for safety arguments require some thought. This would allow automated transformations for refactoring arguments.

7.3.6 Maintaining Consistency between Artifacts

It has gradually become clear that there are numerous consistency (and, more generally, traceability) relations that need to be maintained between the various assurance artifacts: for example, the relation between module views and the corresponding modules, or between a query view and its source diagram. Rather than implementing these maps in an ad hoc way, we need a principled approach. Indeed, the problem has become more acute as we have extended AdvoCATE with additional assurance artifacts, such as first class hazard tables, and bow tie diagrams (not described in this paper; see [28]). We believe that bidirectional transformations [67] offer a rich theoretical foundation that should be applicable here and we have begun to investigate their use in AdvoCATE.

8 Conclusion

We believe that both a formal basis and tool support are crucial for improving the credibility and wider acceptance of structured safety arguments during the certification of safety-critical products. Towards that end, in this paper, we have described

AdvoCATE, a toolset that provides unique automation features to support the development of assurance cases, and a methodology for safety argument development that leverages our toolset. AdvoCATE implements a formal model of graphical argumentation, and the semantics are based on a family of graph-based structures, enriched with metadata. There are orthogonal extensions to define argument patterns, as well as the two key structuring mechanisms, i.e., hierarchy and modularity. Thus, AdvoCATE supports both manual and automated construction of arguments, their modular and hierarchical organization, the specification and instantiation of argument patterns, queries and views on arguments, and the verification of argument properties.

The need for these capabilities was borne out of practical necessity to provide safety case development and safety assurance support for UAS and their missions being conducted at NASA Ames. Indeed, to illustrate the utility of AdvoCATE, and the applicability of its automation features, we have used the tool during the development of assurance arguments for a number of real aviation systems. In particular, we have used the pattern instantiation capability to integrate hazards and requirements analysis [19], as well as evidence and reasoning from formal methods [21, 22], during the development of an end-to-end safety argument for the Swift UAS. Likewise, our colleagues used this capability to create safety arguments from the hazard and safety requirements analysis of a transport-category, twin-engine aircraft model [12].

Additionally, we have applied the query/view mechanism to illustrate the creation of stakeholder specific views on fragments of the Swift UAS safety argument. In particular, we use the AdvoCATE Query Language (AQL) to specify queries, which are applied to arguments augmented with domain-specific metadata, and produce views showing

- traceability between hazards, safety, system, and software requirements;
- whether, and the extent to which, requirements arising from regulations are being addressed;
- the contribution of software to system assurance concerns; and,
- the degree to which an argument is complete [18].

Moreover, we have used the verification environment to specify and verify structural properties on the arguments created. Lastly, we have applied modular structuring to specify an argument architecture, together with modular arguments, for the safety assurance of a ground-based detect and avoid capability used in UAS transit operations. Subsequently, we have applied hierarchical structuring to abstract argument fragments of the same system [27].

We have used AdvoCATE in real projects for safety (and airworthiness) assurance of UAS. Our hypothesis—partly based on this first-hand experience in creating safety cases for real systems, which subsequently underwent regulatory scrutiny—is that formally-founded automation can augment and improve the current practice of manually creating, evolving and evaluating arguments. We plan to evaluate AdvoCATE by application within various other aviation contexts, e.g., airspace system safety, and airworthiness assurance.

Acknowledgements Several individuals have contributed to the development and testing of AdvoCATE. In particular, we thank Josef Pohl, Dwight Naylor (especially for queries), Iain Whiteside (especially

for hierarchy), Atef Suleiman, Alfredo Bencomo, Nija Shi, and Peter Tran. We also acknowledge David Bushnell, Martin Feather, Ibrahim Habli, and Lawrence Markosian for providing end-user feedback.

References

1. Adelard LLP: Assurance and Safety Case Environment (ASCE). `http://www.adelard.com/asce/` (2011)
2. Armengaud, E.: Automated safety case compilation for product-based argumentation. Presented at the 6th European Congress on Embedded Real-time Software and Systems (ERTS^2 2014) (2014)
3. Ayoub, A., Chang, J., Sokolsky, O., Lee, I.: Assessing the Overall Sufficiency of Safety Arguments. In: Proceedings of the 21st Safety-Critical Systems Symposium (SSS '13), pp. 127–144 (2013)
4. Barry, M.R.: CertWare: A workbench for safety case production and analysis. In: Proceedings of the 2011 IEEE Aerospace Conference, pp. 1–10 (2011)
5. Basir, N., Denney, E., Fischer, B.: Constructing a safety case for automatically generated code from formal program verification information. In: M. Harrison, M.A. Sujan (eds.) Computer Safety, Reliability, and Security, *Lecture Notes in Computer Science*, vol. 5219, pp. 249–262. Springer Berlin / Heidelberg (2008)
6. Berthold, R., Denney, E., Fladeland, M., Pai, G., Storms, B., Sumich, M.: Assuring ground-based detect and avoid for UAS operations. In: Proceedings of the 33rd IEEE/AIAA Digital Avionics Systems Conference (DASC), pp. 6A1–1–6A1–16 (2014)
7. Bienvenu, M., ten Cate, B., Lutz, C., Wolter, F.: Ontology-based data access: A study through disjunctive Datalog, CSP, and MMSNP. In: Proceedings of the 32nd Symposium on Principles of Database Systems, pp. 213–224. ACM (2013)
8. Bishop, P., Bloomfield, R.: A methodology for safety case development. In: F. Redmill, T. Anderson (eds.) Industrial Perspectives of Safety-Critical Systems: Proceedings of the 6th Safety-critical Systems Symposium. Springer (1998)
9. Bloomfield, R., Bishop, P.: Safety and Assurance Cases: Past, Present and Possible Future – An Adelard Perspective. In: Proceedings of the 18th Safety-Critical Systems Symposium (2010)
10. Bloomfield, R., Chozos, N., Embrey, D., Henderson, J., Kelly, T., Koornneef, F., Pasquini, A., Pozzi, S., Sujan, M., Cleland, G., Habli, I., Medhurst, J.: Evidence: Using Safety Cases in Industry and Healthcare. The Health Foundation (2012)
11. Blume, M., Appel, A.W.: Hierarchical modularity. ACM Transactions on Programming Languages and Systems **21**, 813–847 (1999)
12. Brat, G., Bushnell, D., Davies, M., Giannakopoulou, D., Howar, F., Kahsai, T.: Verifying the Safety of a Flight-Critical System. arXiv cs.SE e-print 1502.02605 (2015)
13. Clothier, R., Denney, E., Pai, G.: Making a risk informed safety case for small unmanned aircraft system operations. In: Proceedings of the 17th AIAA Aviation Technology, Integration, and Operations Conference (ATIO 2017) (2017)
14. Cruanes, S., Hamon, G., Owre, S., Shankar, N.: Tool integration with the evidential tool bus. In: R. Giacobazzi, J. Berdine, I. Mastroeni (eds.) Verification, Model Checking, and Abstract Interpretation, *Lecture Notes in Computer Science*, vol. 7737, pp. 275–294. Springer Berlin Heidelberg (2013)
15. Denney, E., Habli, I., Pai, G.: Perspectives on Software Safety Case Development for Unmanned Aircraft. In: Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012), pp. 1–8. Boston, MA (2012)
16. Denney, E., Habli, I., Pai, G.: Dynamic safety cases for through-life safety assurance. In: Proceedings of the 37th International Conference on Software Engineering (ICSE 2015): New Ideas and Emerging Results track (NIER). Florence, Italy (2015)
17. Denney, E., Ippolito, C., Lee, R., Pai, G.: An Integrated Safety and Systems Engineering Methodology for Small Unmanned Aircraft Systems. In: Infotech@Aerospace, AIAA 2012-2572. Garden Grove, CA (2012)
18. Denney, E., Naylor, D., Pai, G.: Querying Safety Cases. In: A. Bondavalli, F.D. Giandomenico (eds.) 33rd International Conference on Computer Safety, Reliability and Security (SAFECOMP 2014), pp. 294–309. Springer (2014)
19. Denney, E., Pai, G.: A lightweight methodology for safety case assembly. In: F. Ortmeier, P. Daniel (eds.) Proceedings of the 31st International Conference on Computer Safety, Reliability and Security (SAFECOMP 2012), *LNCS*, vol. 7612, pp. 1–12. Springer-Verlag (2012)

20. Denney, E., Pai, G.: A Formal Basis for Safety Case Patterns. In: F. Bitsch, J. Guiochet, M. Kaâniche (eds.) Computer Safety, Reliability and Security (SAFECOMP 2013), *LNCS*, vol. 8153, pp. 21–32 (2013)
21. Denney, E., Pai, G.: Evidence arguments for using formal methods in software certification. In: 2013 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), pp. 375–380 (2013)
22. Denney, E., Pai, G.: Automating the Assembly of Aviation Safety Cases. IEEE Transactions on Reliability **63**(4), 830–849 (2014)
23. Denney, E., Pai, G.: Safety Case Patterns: Theory and Applications. Technical Report NASA/TM-2015-218492, NASA Ames Research Center (2015)
24. Denney, E., Pai, G.: Architecting a Safety Case for UAS Flight Operations. In: 34th International System Safety Conference (ISSC) (2016)
25. Denney, E., Pai, G., Habli, I.: Towards measurement of confidence in safety cases. In: Proceedings of the 5th International Symposium on Empirical Software Engineering and Measurement, pp. 380–383 (2011)
26. Denney, E., Pai, G., Pohl, J.: AdvoCATE: An Assurance Case Automation Toolset. In: F. Ortmeier, P. Daniel (eds.) SAFECOMP 2012 Workshops—Next Generation of System Assurance Approaches for Safety-Critical Systems (SASSUR), *LNCS*, vol. 7613. Springer-Verlag (2012)
27. Denney, E., Pai, G., Whiteside, I.: Formal foundations for hierarchical safety cases. In: Proceedings of the 16th IEEE International Symposium on High Assurance Systems Engineering (HASE 2015) (2015)
28. Denney, E., Pai, G., Whiteside, I.: Modeling the Safety Architecture of UAS Flight Operations. In: S. Tonetta, E. Schoitsch, F. Bitsch (eds.) Proceedings of the 36th International Conference on Computer Safety, Reliability, and Security (SAFECOMP 2017), Lecture Notes in Computer Science (LNCS). Springer (2017 (to appear))
29. Denney, E., Trac, S.: A software safety certification tool for automatically generated guidance, navigation and control code. In: IEEE Aerospace Conference Electronic Proceedings. IEEE, Big Sky, Montana (2008)
30. Despotou, G., Apostolakis, A., Kolovos, D.: Assuring Dependable and Critical Systems: Implementing the Standards for Assurance Cases with ACedit. White Paper (2012)
31. Dezfuli, H., Benjamin, A., Everett, C., Smith, C., Stamatelatos, M., Youngblood, R.: NASA/SP-2010-580, NASA System Safety Handbook. Volume 1, System Safety Framework and Concepts for Implementation. NASA (2011)
32. Eagles, S., Wu, F.: Reducing Risks and Recalls: Safety Assurance Cases For Medical Devices. Biomedical Instrumentation and Technology **48**(1), 24–32 (2014)
33. European Organisation for the Safety of Air Navigation (EUROCONTROL): Safety Case Development Manual, 2.1 edn. DAP/SSH/091 (2006)
34. European Organisation for the Safety of Air Navigation (EUROCONTROL): Preliminary Safety Case for ADS-B Airport Surface Surveillance Application. PSC ADS-B-APT (2011). URL `http://www.eurocontrol.int/articles/cascade-documents/`
35. Felici, M.: Modeling safety case evolution – examples from the air traffic management domain. In: N. Guelfi, A. Savidis (eds.) Proceedings of the 2nd International Workshop on Rapid Integration of Software Engineering Techniques (RISE), *Lecture Notes in Computer Science*, vol. 3943, pp. 81–96. Springer Berlin Heidelberg (2006)
36. Fenn, J., Hawkins, R., Williams, P., Kelly, T.: Safety Case Composition Using Contracts - Refinements based on Feedback from an Industrial Case Study. In: Proceedings of the 15th Safety Critical Systems Symposium (SSS' 07) (2007)
37. Gacek, A., Backes, J., Cofer, D., Slind, K., Whalen, M.: Resolute: An Assurance Case Language for Architecture Models. In: Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology, HILT '14, pp. 19–28. ACM, New York, NY, USA (2014)
38. Gallina, B.: A model-driven safety certification method for process compliance. In: Proceedings of the 2014 International Symposium on Software Reliability Engineering (ISSRE) Workshops, pp. 204–209 (2014)
39. Goal Structuring Notation Working Group: GSN Community Standard Version 1 (2011). URL `http://www.goalstructuringnotation.info/`
40. Goodenough, J., Weinstock, C., Klein, A.: Eliminative induction: A basis for arguing system confidence. In: Proceedings of the 35th International Conference on Software Engineering (ICSE), pp. 1161–1164 (2013)

41. Graydon, P.: Formal Assurance Arguments: A Solution In Search of a Problem? In: Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) (2015)
42. Graydon, P., Knight, J., Green, M.: Certification and safety cases. In: Proceedings of the 28th International System Safety Conference (2010)
43. Greenwell, W., Knight, J., Holloway, C.M., Pease, J.: A taxonomy of fallacies in system safety arguments. In: Proceedings of the International System Safety Conference (2006)
44. Hawkins, R., Habli, I., Kelly, T.: Principled Construction of Software Safety Cases. In: 2013 SAFECOMP Workshops–Next Generation of System Assurance Approaches for Safety-Critical Systems (SASSUR) (2013)
45. Hawkins, R., Habli, I., Kolovos, D., Paige, R., Kelly, T.: Weaving an assurance case from design: A model-based approach. In: Proceedings of the 16th IEEE International Symposium on High Assurance Systems Engineering (HASE), pp. 110–117 (2015)
46. Hawkins, R., Kelly, T., Knight, J., Graydon, P.: A new approach to creating clear safety arguments. In: Proceedings of the Safety Critical Systems Symposium (2011)
47. International Atomic Energy Agency: IAEA Safety Glossary: Terminology Used in Nuclear Safety and Radiation Protection, 2007 edn. (2007)
48. International Civil Aviation Organization (ICAO) Asia and Pacific Office: Building a Safety Case for Delivery of an ADS-B Separation Service. Guidance Material v1.0 (2011)
49. International Organization for Standardization (ISO): Road Vehicles–Functional Safety. ISO 26262 (2011)
50. Jøsang, A., Bradley, D., Knapskog, S.J.: Belief-based risk analysis. In: Proceedings of the Australasian Information Security Workshop (AISW), pp. 591–598 (2004)
51. Kelly, T.: Arguing safety: A systematic approach to managing safety cases. Ph.D. thesis, University of York (1998)
52. Kelly, T., Bates, S.: The Costs, Benefits, and Risks Associated With Pattern-Based and Modular Safety Case Development. In: Proceedings of the UK MoD Equipment Safety Assurance Symposium (2005)
53. Knight, J., Aiello, A., Hocking, A., Rowanhill, J.: SCT: A Safety Case Toolkit. In: Workshop Proceedings of the 2014 IEEE International Symposium on Software Reliability Engineering (ISSRE)—Assurance Cases for Software-intensive Systems (ASSURE) (2014)
54. Littlewood, B., Wright, D.: The use of multilegged arguments to increase confidence in safety claims for software-based systems: A study based on a BBN analysis of an idealized example. IEEE Transactions on Software Engineering **33**(5), 347–365 (2007)
55. Mahapatra, S.: Automatic report generation in model-based design. SAE Technical Paper 2010-01-2000, SAE International (2010)
56. Matsuno, Y., Takamura, H., Ishikawa, Y.: Dependability case editor with pattern library. In: Proceedings of the 12th IEEE International Symposium on High-Assurance Systems Engineering (HASE), pp. 170–171 (2010)
57. McDermid, J.: Support for safety cases and safety arguments using SAM. Reliability Engineering and System Safety **43**(2), 111–127 (1994)
58. Menon, C., Hawkins, R., McDermid, J.: Interim standard of best practice on software in the context of DS 00-56 Issue 4. Standard of Best Practice Issue 1, Software Systems Engineering Initiative, University of York (2009)
59. Nair, S., Walkinshaw, N., Kelly, T., Vara, J.L.d.l.: An evidential reasoning approach for assessing confidence in safety evidence. Tech. Rep. 2014-17, Simula Research Laboratory (2014)
60. Object Management Group: Structured Assurance Case Metamodel (SACM) version 1.0. Formal/2013-02-01 (2013)
61. Ratiu, D., Zeller, M., Killian, L.: Safety.lab: Model-based domain specific tooling for safety argumentation. In: F. Koornneef, C. van Gulijk (eds.) Proceedings of SAFECOMP 2015 Workshops, pp. 72–82. Springer International Publishing (2015)
62. Ruiz, A., Larrucea, X., Espinoza, H.: A tool suite for assurance cases and evidences: Avionics experiences. In: R.V. O'Connor, M. Umay Akkaya, K. Kemaneci, M. Yilmaz, A. Poth, R. Messnarz (eds.) Systems, Software and Services Process Improvement, *Communications in Computer and Information Science*, vol. 543, pp. 63–71. Springer (2015)
63. Rushby, J.: The interpretation and evaluation of assurance cases. Tech. Rep. SRI-CSL-15-01, Computer Science Laboratory, SRI International, Menlo Park, CA (2015)
64. S-18, Aircraft And System Development And Safety Assessment Committee: ARP 4761, Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equip-

ment. Society of Automotive Engineers (SAE) (1996)
65. S-18, Aircraft And System Development And Safety Assessment Committee: ARP 4754, Guidelines for Development of Civil Aircraft and Systems. Society of Automotive Engineers (SAE) (2010)
66. Steele, P., Collins, K., Knight, J.: ACCESS: A Toolset for Safety Case Creation and Management. In: Proceedings of the 29th International Systems Safety Conference (2011)
67. Stevens, P.: A landscape of bidirectional model transformations. In: Generative and transformational techniques in software engineering II, pp. 408–424. Springer (2008)
68. Taguchi, K., Daisuke, S., Nishihara, H., Takai, T.: Linking traceability with GSN. In: Proceedings of the IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), pp. 192–197 (2014). DOI 10.1109/ISSREW.2014.79
69. Takeyama, M.: A note on D-Cases as proofs as programs. Tech. rep., National Institute of Advanced Industrial Science and Technology, Osaka, Japan (2010). AIST-PS-2010-007
70. Toulmin, S.E.: The Uses of Argument. Cambridge University Press (1969)
71. UK Civil Aviation Authority (CAA): Small Unmanned Aircraft: Congested Areas Operating Safety Case (CAOSC). Information Notice IN-2014/184 (2014)
72. UK Ministry of Defence (MOD): Safety Management Requirements for Defence Systems (2007)
73. UK Ministry of Defence (MOD): The 'White Booklet': An Introduction to System Safety Management in the MOD. Issue 3 (2011)
74. UK Rail Safety Standards Board: Engineering Safety Management. Issue 4 (2007)
75. US Department of Transportation, Federal Aviation Administration (FAA): Software Approval Guidelines. FAA Order 8110.49 Chg 1 (2011)
76. US Department of Transportation, Federal Aviation Administration (FAA): Unmanned Aircraft Systems (UAS) Operational Approval. National Policy N 8900.227 (2013)
77. US Department of Transportation, Federal Aviation Administration (FAA): Flight Standards Information Management System, Volume 16, Unmanned Aircraft Systems. Order 8900.1 (2014)
78. US Food and Drug Administration (FDA): Guidance for Industry and FDA Staff - Total Product Life Cycle: Infusion Pump - Premarket Notification [510(k)] Submissions (2010)
79. Voss, S., Sch$^{\acute{}}$atz, B., Khalil, M., Carlan, C.: Towards modular certification using integrated model-based safety cases. In: Proceedings of the 25th International Conference on Computer Aided Verification (CAV), Workshop on Assurance and Verification (VeriSure 2013) (2013)
80. Wassyng, A., Maibaum, T., Lawford, M., Bherer, H.: Software certification: Is there a case against safety cases? In: Foundations of Computer Software, Modeling, Development and Verification of Adaptive Systems, *Lecture Notes in Computer Science*, vol. 6662, pp. 206–227. Springer-Verlag (2011)
81. Weinstock, C.B., Goodenough, J.B., Klein, A.Z.: Measuring Assurance Case Confidence Using Baconian Probabilities. In: Proceedings of the 1st International Workshop on Assurance Cases for Software-Intensive Systems, ASSURE '13, pp. 7–11. IEEE Press (2013)
82. Wilson, S., McDermid, J., Kirkham, P., Fenelon, P.: The Safety Argument Manager: An integrated approach to the engineering and safety assessment of computer-based systems. In: Proceedings of the IEEE Symposium and Workshop on Engineering of Computer-Based Systems., pp. 198–205 (1996). DOI 10.1109/ECBS.1996.494529
83. Yang, J.B., Xu, D.L.: On the evidential reasoning algorithm for multiple attribute decision analysis under uncertainty. IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans **32**(3), 289–304 (2002)
84. Yuan, T., Kelly, T., Xu, T., Wang, H., Zhao, L.: A Dialogue-based Safety Argument Review Tool. In: Proceedings of the 1st International Workshop on Argument for Agreement and Assurance (AAA-2013) (2013)